TABLE OF CONTENTS

Introduction ........................................................................................................................................... 2
Memory Address Translation ..................................................................................................................... 2
  Quick Translation Formula .................................................................................................................... 2
  Smart Translation Formula .................................................................................................................. 2
Symbols .................................................................................................................................................. 3
  Fat Header ........................................................................................................................................ 4
  Mach Header ..................................................................................................................................... 5
Information Extraction (Also Know As Analysis) .................................................................................... 7
  Machine Information .......................................................................................................................... 7
  Mounted File Systems ....................................................................................................................... 7
  BSD Processes .................................................................................................................................. 10
  Kernel Extensions (Also Known As Drivers, Kernel Modules) ............................................................ 15
  System Calls ..................................................................................................................................... 16
Thanks .................................................................................................................................................. 18

[ADVANCED MAC OS X PHYSICAL MEMORY ANALYSIS]

In 2008 and 2009, companies and governments (e.g. Law Enforcement agencies) interests for Microsoft Windows physical memory grew significantly. Now it is time to talk about Mac OS X. This paper will introduce basis of Mac OS X Kernel Internals regarding management of processes, threads, files, system calls, kernel extensions and more. Moreover, we are going to details how to initialize and perform a virtual to physical translation under an x86 Mac OS X environment.
ADVANCED MAC OS X PHYSICAL MEMORY ANALYSIS

INTRODUCTION

In 2008 and 2009, companies and governments (e.g. Law Enforcement agencies) interests for Microsoft Windows physical memory grew significantly. Now it is time to talk about Mac OS X. This paper introduces Mac OS X Kernel Internals regarding management of processes, threads, files, system calls, kernel extensions and more. We provide details on how to initialize and perform a virtual to physical translation under a x86 Mac OS X environment.

Physical Memory is widely known in the UNIX world as /dev/mem.

MEMORY ADDRESS TRANSLATION

QUICK TRANSLATION FORMULA

Most Operating Systems have a way to compute the kernel physical address even if you do not have the cr3 register value which is used as Directory Table Base for virtual to physical address translation. If you want to have more detailed information on this, please refer to Intel64 and IA-32 Architectures Software Developer’s Manual: Volume 3A System Programming Guide.\(^1\)

By kernel physical addresses, I mean the kernel image (\_DATA & \_CODE sections) physical address. Both contain important information and variables we need. For instance, to reconstruct the kernel address space we need to be able to use Smart Translation Formula which requires variables we can retrieve using Quick Translation Formula. As I said above, with Quick Translation Formula we can only access to \_DATA and \_CODE sections of the kernel image and not to allocated buffers.

Here is a summary of some operating systems with their corresponding formula to translate from Kernel Virtual Address (KVA) to Kernel Physical Address (KPA).

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Quick translation formula</th>
</tr>
</thead>
<tbody>
<tr>
<td>x86 Linux</td>
<td>KPA = KVA – 0xC0000000</td>
</tr>
<tr>
<td>PlayStation 3 Linux</td>
<td>KPA = KVA - 0xC00000000000000000000000000000000</td>
</tr>
<tr>
<td>x86 Windows</td>
<td>KPA = KVA &amp; 0x1FFFF000</td>
</tr>
<tr>
<td>Mac OS X</td>
<td>KPA = KVA</td>
</tr>
</tbody>
</table>

As you can see the formula for Mac OS X, is the easiest existing formula.

SMART TRANSLATION FORMULA

Using Quick Translation Formula, we can retrieve variables from \_DATA section and initialized by slave_pstart() function of Mac OS X Kernel, which is called during the Operating System initialization.

---

\(^1\) 3.6 PAGING (VIRTUAL MEMORY) OVERVIEW.
There are ~4 variables which are interesting to perform the Smart Translation Formula: IdlePDPT, IdlePDPT64, IdlePML4 and IdlePTD.

IdlePML4 variable is initialized even on 32-bits Operating System. PML4 stands for Page Map Level 4 paging structure. This method can be used to address up to $2^{27}$ pages, which spans a linear address space of $2^{48}$ bytes.

Then, using IdlePML4 variable we can cover a translation mechanism for a linear address space of $2^{48}$ bytes even if the processor cannot do it. Internally, in kernel structures, Mac OS X is using 64-bits addressing for memory objects.

These variable are used later to initialize kernel_map and kernel_pmap kernel structures/variables.

Here is a common output of these variables under Mac OS X Leopard.

* _IdlePML4: [0x004EB00C] = 0x01219000
  0x01219000: 27 A0 21 01 00 00 00 00 - 00 00 00 00 00 00 00 00 ................
  0x01219010: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 ................
  0x01219020: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 ................
  0x01219030: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 ................
  0x01219040: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 ................
  0x01219050: ................

* _IdlePDPT64: [0x004EB010] = 0x0121A000
  0x0121A000: 27 C0 21 01 00 00 00 00 - 27 D0 21 01 00 00 00 00 ................
  0x0121A010: 27 E0 21 01 00 00 00 - 27 F0 21 01 00 00 00 00 ................
  0x0121A020: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 ................
  0x0121A030: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 ................
  0x0121A040: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 ................
  0x0121A050: ................

* _IdlePDPT: [0x004EB008] = 0x0121B000
  0x0121B000: 01 C0 21 01 00 00 00 00 - 01 D0 21 01 00 00 00 00 ................
  0x0121B010: 01 E0 21 01 00 00 00 00 - 01 F0 21 01 00 00 00 00 ................
  0x0121B020: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 ................
  0x0121B030: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 ................
  0x0121B040: 00 00 00 00 00 00 00 00 - 00 00 00 00 00 00 00 00 ................
  0x0121B050: ................

* _IdlePTD: [0x004EB004] = 0x0121C000
  0x0121C000: 63 50 02 01 00 00 00 00 - 63 60 02 01 00 00 00 00 cF........c`......
  0x0121C010: 63 70 02 01 00 00 00 00 - 63 80 02 01 00 00 00 00 cp........c......
  0x0121C020: 23 90 02 01 00 00 00 00 - 23 A0 02 01 00 00 00 00 ................
  0x0121C030: 63 B0 02 01 00 00 00 00 - 63 C0 02 01 00 00 00 00 c........c......
  0x0121C040: 63 D0 02 01 00 00 00 00 - 63 E0 02 01 00 00 00 00 c........c......
  0x0121C050: 63 F0 02 01 00 00 00 00 - 63 00 03 01 00 00 00 00 c........c......

**SYMBOLS**

Symbols are a key element of volatile memory forensics without them an advanced analysis is impossible. Symbols of Microsoft Windows are available on a remote server as standalone files, but on Mac OS X symbols are directly stored inside the executable in a segment/section called __LINKEDIT.
The easiest way to retrieve kernel symbols is to extract them from the kernel executable of the hard-drive.

Symbols are firstly used to retrieve the address of memory variable for Smart Translation Formula.

**FAT HEADER**

Mac OS X file format follows the FAT file format which contains magic signature of the header and the number of different architectures entries (i386, PowerPC or Both) inside the executable in big endian.

```c
#define FAT_MAGIC 0xBEBAFECA

typedef struct _FAT_HEADER
{
    ULONG magic;
    ULONG nfat_arch;
} FAT_HEADER, *PFAT_HEADER;
```

To jump to the first architecture entry we add `sizeof(FAT_HEADER)` bytes to the pointer of the file header. Each entry uses the following definition, and also uses the big endian endianess.

```c
typedef struct _FAT_ARCH
{
    cpu_type_t cputype;
    cpu_subtype_t cpusubtype;
    ULONG offset;
    ULONG size;
    ULONG align;
} FAT_ARCH, *PFAT_ARCH;
```

The first field, `cpu_type`, indicates to the loader what kind of architecture this entry defines using the following description:

```c
typedef enum
{
    CPU_TYPE_VAX = 1,
    CPU_TYPE_ROMP = 2,
    CPU_TYPE_NS32032 = 4,
    CPU_TYPE_NS32332 = 5,
    CPU_TYPE_MC680x0 = 6,
    CPU_TYPE_I386 = 7,
    CPU_TYPE_MIPS = 8,
    CPU_TYPE_NS32532 = 9,
    CPU_TYPE_MC98000 = 10,
    CPU_TYPE_HPPA = 11,
    CPU_TYPE_ARM = 12,
    CPU_TYPE_MC88000 = 13,
    CPU_TYPE_SPARC = 14,
    CPU_TYPE_I860 = 15,
    CPU_TYPE_ALPHA = 16,
    CPU_TYPE_POWERPC = 18,
    /* APPLE LOCAL 64-bit */
    CPU_TYPE_POWERPC_64 = (18 | CPU_IS64BIT),
    /* APPLE LOCAL x86_64 */
    CPU_TYPE_X86_64 = (CPU_TYPE_I386 | CPU_IS64BIT)
} cpu_type_t;
```
And the third field, offset, contains the raw offset of the architecture header.

We assume index x is the id of the CPU_TYPE_I386 architecture. So we have FAT_ARCH[x].cputype equals to CPU_TYPE_I386 and FAT_ARCH[x].offset as new pointer offset to the MACH_HEADER structure.

MACH_HEADER
Now we have a pointer the i386 architecture binary using the following header definition and little-endian endianness.

```c
#define MH_MAGIC 0xfeedface

typedef struct _MACH_HEADER {
    ULONG Magic;
    cpu_type_t cputype;
    cpu_subtype_t cpusubtype;
    ULONG filetype;
    ULONG ncmds;
    ULONG sizeofcmds;
    ULONG flags;
} MACH_HEADER, *PMACH_HEADER;
```

This architecture validity can be verified using the 0xfeedface magic key.

Now we can read what Apple calls commands, the field MACH_HEADER.ncmds indicates the number of commands inside the Mach-O binary.

We have to add sizeof(MACH_HEADER) to the Mach-O header pointer to have a pointer to the first command entry. There are different commands types and size of commands depends of their type. Most important commands types are LC_SEGMENT and LC_SYMTAB.

```c
#define LC_SEGMENT 0x1 /* file segment to be mapped */
#define LC_SYMTAB  0x2 /* link-edit stab symbol table info (obsolete) */
```

And very two first fields contains information about the command’s type and its size, using the following scheme:

```c
typedef struct _LOAD_COMMAND {
    ULONG  cmd; /* type of load command */
    ULONG  cmdsize; /* total size of command in bytes */
} LOAD_COMMAND, *PLOAD_COMMAND;
```

Command type called LC_SYMTAB, contains raw pointers to two different tables. One, called symoff, with NLIST structures-based entries, and another, called stroff, with functions and variables names of each corresponding entry in the same order.

```c
typedef struct _SYMTAB_COMMAND {
    ULONG cmd;
    ULONG cmdsize;
} SYMTAB_COMMAND;
```
Both `symoff` and `stroff` are pointer into the __LINKEDIT segment. Please note we have to add `FAT_ARCH[x].offset` value to these fields. And `n_value` field from NLIST structure contains the symbol offset.

Here is a short dump of symbols retrieved from Mac OS X Leopard kernel.

```plaintext
[000000] .constructors_used 0x0050F254
[000001] .destructors_used 0x0050F25C
[000002] _AARPwakeup 0x0029F6BD
[000003] _APTD 0xFF7F8000
[000004] _APTDpde 0xFEFF7FC0
[000005] _APTmap 0xFF000000
[000006] _ASPgetmsg 0x00293448
[000007] _ASPputmsg 0x00293A3F
[000008] _ATPgetreq 0x002A5A12
[000009] _ATPgetrsp 0x002A5A79
[000010] _ATPsndreq 0x002A592C
[000011] _ATPsndrsp 0x002A599F
[000012] _ATgetmsg 0x002A5844
[000013] _ATputmsg 0x002A58B8
[..]
[000223] _IOZeroTvalspec 0x0050EE18
[000224] _IS_64BIT_PROCESS 0x00373952
[000225] _IdlePDPT 0x004EB008
[000226] _IdlePDPT64 0x004EB010
[000227] _IdlePML4 0x004EB00C
[000228] _IdlePTD 0x004EB004
[000229] _InitGlobals 0x002A3A08
[000230] _InsertKeyRecord 0x003477C8
[000231] _InsertOffset 0x003475A1
[000232] _InsertRecord 0x00347703
[..]
[002577] __ZN32IOServiceMessageUserNotificationC2EPK11OSMetaClass 0x004354AA
[002578] __ZN32IOServiceMessageUserNotificationC2Ev 0x0043550E
[002579] __ZN32IOServiceMessageUserNotificationD0Ev 0x0043553A
[002580] __ZN32IOServiceMessageUserNotificationD1Ev 0x0043551A
[002581] __ZN32IOServiceMessageUserNotificationD2Ev 0x0043554A
[002582] __ZN510CPU10gMetaClassE 0x0052E838
[002583] __ZN510CPU10superClassE 0x004BE7C
[002584] __ZN510CPU11getCPUTagEv 0x004300A2
[002585] __ZN510CPU11getCPUTagEv 0x00430082
[002586] __ZN510CPU11setCPUTagEv 0x0043008E
[002587] __ZN510CPU11setPropertyEPK8OSSymbolP8OSObject 0x0042FE68
[..]
```
INFORMATION EXTRACTION (ALSO KNOWN AS ANALYSIS)

Once memory manager is functional, we can now proceed to the extraction of information such as process list and so on.

MACHINE INFORMATION

Machine identification is a very important part to validate result. This section covers how to retrieve Darwin version, compilation date, number of CPUs and available memory on the current system.

There is a global variable, accessible from symbols, called `version` which contains a 100 bytes string with O.S. Type, O.S. Release version, username who compiled it.

There is another global variable, accessible from symbols, called `machine_info` defined by `machine_info` structure which contains information about CPUs and Memory of the target machine.

Definition of `machine_info` structure can be retrieved in `xnu/osfmk/mach/machine.h` header file.

Below is the definition of `machine_info` structure under Mac OS X Snow Leopard.

```c
struct machine_info {
    integer_t major_version; /* kernel major version id */
    integer_t minor_version; /* kernel minor version id */
    integer_t max_cpus; /* max number of CPUs possible */
    uint32_t memory_size; /* size of memory in bytes, capped at 2 GB */
    uint64_t max_mem; /* actual size of physical memory */
    uint32_t physical_cpu; /* number of physical CPUs now available */
    integer_t physical_cpu_max; /* max number of physical CPUs possible */
    uint32_t logical_cpu; /* number of logical CPU now available */
    integer_t logical_cpu_max; /* max number of physical CPUs possible */
};
```

Above is a screenshot of extraction information showing the target machine is running Mac OS X Leopard 10.5.0 with 1GB of physical memory.

MOUNTED FILE SYSTEMS
Mounted file systems are defined by a global list-head, accessible from symbols, called `mountlist`. `mountlist` is a single link-list and contains a pointer called next which is a pointer to the next mounted file system entry both are defined by `mount` structure.

This structure contains 3 important fields including: file system type (`f_fstypename`), directory on which mounted (`f_mntonname`) and mounted file system (`f_mntfromname`).

Definition of `mount` structure can be retrieved in `xnu/bsd/sys/mount_internal.h` header file.

Below is the definition of `mount` structure under Mac OS X Snow Leopard.

```c
/*
 * Structure per mounted file system. Each mounted file system has an
 * array of operations and an instance record. The file systems are
 * put on a doubly linked list.
 */

struct mount {
    TAILQ_ENTRY(mount) mnt_list;    /* mount list */
    int32_t mnt_count;              /* reference on the mount */
    lck_mtx_t mnt_mlock;            /* mutex that protects mount point */
    struct vfsops *mnt_op;          /* operations on fs */
    struct vfstable *mnt_vtable;   /* configuration info */
    struct vnode *mnt_vnodecovered; /* vnode we mounted on */
    struct vnode *mnt_vnodecovered; /* list of vnodes this mount */
    struct vnode *mnt_vnodecovered; /* list of vnodes this mount */
    struct vnode *mnt_vnodecovered; /* list of vnodes this mount */
    uint32_t mnt_flag;              /* flags */
    uint32_t mnt_kern_flag;         /* kernel only flags */
    uint32_t mnt_lflag;             /* mount life cycle flags */
    uint32_t mnt_maxsymlinklen;     /* max size of short symlink */
    struct vfsstatfs mnt_vfsstat;   /* cache of filesystem stats */
    qaddr_t mnt_data;              /* private data */
    /* Cached values of the IO constraints for the device */
    uint32_t mnt_maxreadcnt;        /* Max. byte count for read */
    uint32_t mnt_maxwritecnt;       /* Max. byte count for write */
    uint32_t mnt_maxreadcnt;        /* Max. segment count for read */
    uint32_t mnt_maxwritecnt;       /* Max. segment count for write */
    uint32_t mnt_maxsegreadsize;    /* Max. segment read size */
    uint32_t mnt_maxsegwrite-size;  /* Max. segment write size */
    uint32_t mnt_alignmentmask;     /* Mask of bits that aren't addressable via DMA */
    uint32_t mnt_devblocksize;      /* the underlying device block size */
    uint32_t mnt_ioqueue_depth;     /* the maximum number of commands a device can accept */
    uint32_t mnt_ioscale;           /* scale the various throttles/limits imposed on the amount of I/O in flight */
    uint32_t mnt_ioflags;           /* flags for underlying device */
    pending_io_t mnt_pending_write_size; /* byte count of pending writes */
    pending_io_t mnt_pending_read_size; /* byte count of pending reads */
    lck_rwlock_t mnt_rwlock;        /* mutex readwrite lock */
    lck_mtx_t mnt_renamelock;       /* mutex that serializes renames that change shape of tree */
    vnode_t mnt_devvp;              /* the device mounted on for local file systems */
}
```
uint32_t mnt_devbsdunit; /* the BSD unit number of the device */
void *mnt_throttle_info; /* used by the throttle code */
int32_t mnt_crossref; /* references to cover lookups crossing into mp */
int32_t mnt_iterref; /* references to cover iterations; drained makes it -ve */

/* XXX 3762912 hack to support HFS filesystem 'owner' */
uid_t mnt_fsowner;
gid_t mnt_fsgroup;

struct label *mnt_mntlabel; /* MAC mount label */
struct label *mnt_fslabel; /* MAC default fs label */

/* cache the rootvp of the last mount point
 * in the chain in the mount struct pointed
 * to by the vnode sitting in '/'
 * this cache is used to shortcircuit the
 * mount chain traversal and allows us
 * to traverse to the true underlying rootvp
 * in 1 easy step inside of 'cache_lookup_path'
 * make sure to validate against the cached vid
 * in case the rootvp gets stolen away since
 * we don't take an explicit long term reference
 * on it when we mount it */
vnode_t mnt_realrootvp;
uint32_t mnt_realrootvp_vid;
/* bumped each time a mount or unmount
 * occurs... its used to invalidate
 * 'mnt_realrootvp' from the cache */
uint32_t mnt_generation;
/*
 * if 'MNTK_AUTH_CACHE_TIMEOUT' is
 * set, then 'mnt_authcache_ttl' is
 * the time-to-live for the per-vnode authentication cache
 * on this mount... if zero, no cache is maintained...
 * if 'MNTK_AUTH_CACHE_TIMEOUT' isn't set, its the
 * time-to-live for the cached lookup right for
 * volumes marked 'MNTK_AUTH_OPAQUE'.
 */
int mnt_authcache_ttl;
/*
 * The proc structure pointer and process ID form a
 * sufficiently unique duple identifying the process
 * hosting this mount point. Set by vfs_markdependency()
 * and utilized in new_vnode() to avoid reclaiming vnodes
 * with this dependency (radar 5192010).
 */
pid_t mnt_dependent_pid;
void *mnt_dependent_process;
};
Above is a screenshot of mounted file systems including an external hard-drive.

**BSD Processes**

Every Operating System uses user-land processes, it is one of the key element of a working O.S.

Loaded processes are stored into `proc` structure which contains a double-list to walk into the list. There is a global variable, retrievable from symbols, called `kernproc` is the list-head of BSD processes list.

`p_list` field is a double link-list which contains a pointer to both, the previous and the next process.

Definition of `proc` structure can be retrieved in `xnu/bsd/sys/proc_internal.h` header file.

Below is the definition of `proc` structure under Mac OS X Snow Leopard.

```c
/*
 * Description of a process.
 *
 * This structure contains the information needed to manage a thread of
 * control, known in UN*X as a process; it has references to substructures
 * containing descriptions of things that the process uses, but may share
 * with related processes. The process structure and the substructures
 * are always addressible except for those marked "(PROC ONLY)" below,
 * which might be addressible only on a processor on which the process
 * is running.
 */

struct proc {
    LIST_ENTRY(proc) p_list;  /* List of all processes. */
    pid_t    p_pid;           /* Process identifier. (static)*/
    void *   task;            /* corresponding task (static)*/
    struct   proc *p_pptr;    /* Pointer to parent process. (LL) */
    pid_t    p_ppid;          /* process's parent pid number */
    pid_t    p_pgrpid;        /* process group id of the process (LL)*/
    lck_mtx_t p_mlock;        /* mutex lock for proc */
    char     p_stat;          /* S* process status. (PL)*/
    char     p_shutdownstate; /* P_KDEBUG eq (CC)*/
    char     p_kdebug;        /* P_BTRACE eq (CC)*/
    LIST_ENTRY(proc) p_pglist; /* List of processes in pgrp. (PGL) */
    LIST_ENTRY(proc) p_sibling; /* List of sibling processes. (LL)*/
    LIST_HEAD(, uthread) p_uthlist; /* Pointer to list of children. (LL)*/
};
```
LIST_ENTRY(proc)  p_hash;  /* Hash chain. (LL) */
TAILQ_HEAD(eventqelt)  p_evlist;  /* (PL) */

lck_mtx_t  p_fdmlock;  /* proc lock to protect fdesc */

/* substructures: */
kauth_cred_t  p_ucred;  /* Process owner's identity. (PL) */
struct  filedesc *p_fd;  /* P' flag to open files structure. (PFDL) */
struct  pstats *p_stats;  /* Accounting/statistics (PL). */
struct  plimit *p_limit;  /* Process limits.(PL) */

struct sigacts *p_sigacts;  /* Signal actions, stat (PL) */
int  p_siglist;  /* signals captured back from threads */
lck_spin_t  p_slock;  /* spin lock for itimer/profil protection */

#define  p_rlimit  p_limit->pl_rlimit

struct  plimit *p_limit;  /* old process limits - not inherited by child (PL) */
unsigned int  p_flag;  /* P' flags. (atomic bit ops) */
unsigned int  p_lflag;  /* local flags (PL) */
unsigned int  p_listflag;  /* list flags (LL) */
unsigned int  p_ladvflag;  /* local adv flags (atomic) */
int  p_refcount;  /* number of outstanding users(LL) */
int  p_childrencnt;  /* children holding ref on parent (LL) */
int  p_parentref;  /* children lookup ref on parent (LL) */

pid_t  p_oppid;  /* Save parent pid during ptrace. XXX */
u_int  p_xstat;  /* Exit status for wait; also stop signal. */

ifdef  _PROC_HAS_SCHEDINFO_
/* may need cleanup, not used */
short_t  p_estcpu;  /* Time averaged value of p_cpticks.(used by aio and */
proc_comapre) */
fixpt_t  p_pctcpu;  /* %cpu for this process during p_swtime (used by */
aio)*/
u_int  p_slptime;  /* used by proc_compare */
endif  /* _PROC_HAS_SCHEDINFO_ */

struct  itimerval  p_realtimer;  /* Alarm timer. (PSL) */
struct  timeval  p_rtime;  /* Real time. (PSL) */
struct  itimerval  p_vtimer_user;  /* Virtual timers. (PSL) */
struct  itimerval  p_vtimer_prof;  /* (PSL) */

struct  timeval  p_rlim_cpu;  /* Remaining rlim cpu value. (PSL) */
int  p_debugger;  /* NU 1: can exec set-bit cpu.(used by aio and */
boolean_tsigwait;  /* indication to suspend (PL) */
void  *sigwait_thread;  /* 'thread' holding sigwait(PL) */
void  *exit_thread;  /* Which thread is exiting(PL) */
int  p_vforkcnt;  /* number of outstanding vforks(PL) */
void  *p_vforkact;  /* activation running this vfork proc)(static) */
int  p_fpdrainwait;  /* (PFDL) */
pid_t  p_contproc;  /* last PID to send us a SIGCONT (PL) */

/* Following fields are info from SIGCHLD (PL) */

pid_t  si_pid;  /* (PL) */
Information Extraction (Also Known As Analysis) | NFI

```c
u_int  si_status; /* (PL) */
u_int  si_code;  /* (PL) */
uid_t  si_uid;    /* (PL) */

void * vm_shm;  /* (SYSV SHM Lock) for sysV shared memory */

#if CONFIG_DTRACE
user_addr_t p_dtrace_argv; /* (write once, read only after that) */
user_addr_t p_dtrace_envp; /* (write once, read only after that) */
lck_mtx_t _p_dtrace_splock; /* sun proc lock emulation */
int _p_dtrace_probes;  /* (PL) are there probes for this proc? */
u_int _p_dtrace_count; /* (sprlock) number of DTrace tracepoints */
struct dtrace_ptss_page* p_dtrace_ptss_pages; /* (sprlock) list of user ptss pages */
struct dtrace_ptss_page_entry* p_dtrace_ptss_free_list; /* (atomic) list of individual ptss entries */
struct dtrace_helpers* p_dtrace_helpers; /* (dtrace_lock) DTrace per-proc private */
struct dof_ioctl_data*p_dtrace_lazy_dofs; /* (sprlock) unloaded dof_helper_t's */
#endif /* CONFIG_DTRACE */

/* XXXXXXXXXXXXX BCOPY'ed on fork XXXXXXXXXXXXXXXX */
/* The following fields are all copied upon creation in fork. */
#define p_startcopy p_argslen

u_int p_argslen; /* Length of process arguments. */
int  p_argc;    /* saved argc for sysctl_procargs() */
user_addr_t user_stack; /* where user stack was allocated */
struct vnode *p_textvp; /* Vnode of executable. */
off_t p_textoff; /* offset in executable vnode */
sigset_t p_sigmask; /* DEPRECATED */
sigset_t p_sigignore; /* Signals being ignored. (PL) */
sigset_t p_sigcatch; /* Signals being caught by user. (PL) */

u_char  p_priority; /* (NU) Process priority. */
u_char  p_resv0;  /* (NU) User-priority based on p_cpu and p_nice. */
char    p_nice;   /* Process "nice" value. (PL) */
char    p_resv1;  /* (NU) User-priority based on p_cpu and p_nice. */

#if CONFIG_MACF
int    p_mac_enforce; /* MAC policy enforcement control */
#endif

char    p_comm[MAXCOMLEN+1];
char    p_name[(2*MAXCOMLEN)+1]; /* PL */

struct  pgrp *p_pgrp;  /* Pointer to process group. (LL) */
int     p_iopol_disk; /* disk I/O policy (PL) */
uint32_t p_csflags; /* flags for codesign (PL) */
uint32_t p_pcaction; /* action for process control on starvation */
uint8_t p_uuid[16];   /* from LC_UUID load command */

/* End area that is copied on creation. */
/* XXXXXXXXXXXXXX End of BCOPY'ed on fork (AIOLOCK)XXXXXXXXXXXXXXXX */
```
#define p_endcopy  p_aio_total_count
     int     p_aio_total_count; /* all allocated AIO requests for this proc */
     int     p_aio_active_count; /* all unfinished AIO requests for this proc */
     TAILQ_HEAD( , aio_workq_entry ) p_aio_activeq; /* active async IO requests */
     TAILQ_HEAD( , aio_workq_entry ) p_aio_doneq; /* completed async IO requests */

struct klist p_klist; /* knote list (PL?) */
struct 
    rusage *p_ru; /* Exit information. (PL) */
thread_t p_signalholder;
thread_t p_transholder;

/* DEPRECATED following field */
uint8_t p_acflag; /* Accounting flags. */

struct lctx *p_lctx; /* Pointer to login context. */
LIST_ENTRY(proc) p_lclist; /* List of processes in lctx. */
user_addr_t p_threadstart; /* pthread start fn */
user_addr_t p_wqthread; /* pthread workqueue fn */
int p_pthsize; /* pthread size */
user_addr_t p_targconc; /* target concurrency ptr */
void * p_wqptr; /* workq ptr */
int p_wqsize; /* allocated size */
boolean_t p_wqiniting; /* semaphore to serialize wg_open */
lck_spin_t p_wqlock; /* lock to protect work queue */
struct timeval p_start; /* starting time */
void * p_rcall;
int p_ractive;
int p_idversion; /* version of process identity */
void * p_pthhash; /* pthread waitqueue hash */

#if DIAGNOSTIC
    unsigned int p_fdlock_pc[4];
    unsigned int p_fdunlock_pc[4];
#endif /* DIAGNOSTIC */

#if SIGNAL_DEBUG
    unsigned int lockpc[8];
    unsigned int unlockpc[8];
#endif /* SIGNAL_DEBUG */

/* DIAGNOSTIC */
uint64_t p_dispatchqueue_offset;
}

Pointer to the process group, pgrp structure, allows us to retrieve the username of the person who launched the program because this structure contains a pointer to a structure called session with the username.

Definition of pgrp structure can also be retrieved in xnu/bsd/sys/proc_internal.h header file.

Below is the definition of pgrp structure under Mac OS X Snow Leopard.

/*
 * One structure allocated per process group.
 */
struct pgrp {


/* Hash chain. (LL) */
LIST_ENTRY(pgrp) pg_hash;

/* Pointer to pgrp members. (PGL) */
LIST_HEAD(, proc) pg_members;

/* Pointer to session. (LL ) */
struct session *pg_session;

/* Pgrp id. (static) */
pid_t pg_id;

/* # procs qualifying pgrp for job control (PGL) */
int pg_jobc;

/* Number of processes in the process group (PGL) */
int pg_membercnt;

/* number of current iterators (LL) */
unsigned int pg_refcount;

/* number of current iterators (LL) */
unsigned int pg_listflags;

/* mutex lock to protect pgrp */
lck_mtx_t pg_mlock;

};

Definition of session structure can also be retrieved in xnu/bsd/sys/proc_internal.h header file.

Below is the definition of session structure under Mac OS X Snow Leopard.

/*
 * One structure allocated per session.
 */
struct session {
    int s_count; /* Ref cnt; pgrps in session. (LL) */
    struct proc *s_leader; /* Session leader.(static) */
    struct vnode *s_ttyvp; /* Vnode of controlling terminal.(SL) */
    int s_ttyvid; /* Vnode id of the controlling terminal (SL) */
    struct tty *s_ttyp; /* Controlling terminal. (SL + ttyvp != NULL) */
    pid_t s_ttypgrpid; /* tty's pgrp id */
    pid_t s_sid; /* Session ID (static) */
    char s_login[MAXLOGNAME]; /* Setlogin() name.(SL) */
    int s_flags; /* Session flags (s_mlock) */
    LIST_ENTRY(session) s_hash; /* Hash chain.(LL) */
    lck_mtx_t s_mlock; /* mutex lock to protect session */
    int s_listflags;
};

s_login field contains the name of the username in ASCII.
Above is a sample screenshot of a processes list, mainly executed by nfinfi user around March 2009.

**KERNEL EXTENSIONS (ALSO KNOWN AS DRIVERS, KERNEL MODULES)**

Kernel-Mode, the God Mode, is the most privileged level of an Operating System. Loaded Kernel Extensions can be retrieved by a global list-head variable, accessible from symbols, called kmod defined by kmod_info structure. The next field points to the next kernel extension.

Definition of session structure can also be retrieved in xnu/osfmk/mach/kmod.h header file.

Below is the definition of kmod_info structure under Mac OS X Snow Leopard.

```c
typedef struct kmod_info {
    struct kmod_info *next;
    int info_version; // version of this structure
    int id;
    char name[KMOD_MAX_NAME];
    char version[KMOD_MAX_NAME];
    int reference_count; // # refs to this
    kmod_reference_t *reference_list; // who this refs
    vm_address_t address; // starting address
    vm_size_t size; // total size
    vm_size_t hdr_size; // unwired hdr size
    kmod_start_func_t *start;
    kmod_stop_func_t *stop;
} kmod_info_t;
```

As you can see here we have both kernel extensions image base start and size. Since we have a functional kernel address space, we can easily extract the image of the kernel extension.
Above is a screenshot of a loaded kext lists.

**SYSTEM CALLS**

The very first step is to localize the syscall table, called *sysent*, which is a non accessible variable from symbols. So using a magic trick we can retrieve its offset through *nsysent* exported variable which contains the number of syscall entries.

Under Mac OS X Leopard (10.5), as explained by Jesse D'Aguanno at BH US 2008, we have to add 0x20 to *nsysent* offset to obtain the offset of syscall table.

Under Mac OS X Snow Leopard (10.6), we have to proceed with a different methodology. First, we have to retrieve the value of *nsysent* variable, then we multiply its value with the size of *sysent* structure, and then we subtract this value to *nsysent* offset to obtain the offset of *sysent* table.

Definition of *sysent* structure can also be retrieved in xnu/bsd/sys/sysent.h header file.

Below is the definition of *sysent* structure under Mac OS X Snow Leopard.

```c
struct sysent {
    /* system call table */
    int16_t sy_narg;  /**< number of args */
    int8_t sy_resv;  /**< reserved */
    int8_t sy_flags;  /**< flags */
    sy_call_t *sy_call;  /**< implementing function */
    sy_munge_t *sy_arg_munge32;  /**< system call arguments munger for 32-bit process */
};
```
Above is a picture showing a list of syscalls from `sysent` table.
Integrity checks are done if entry value does not give the function name value. It does not sound complicated but this trick was enough to detect Jesse D’Aguanno Rootkit presented at HAR2009.

THANKS

For help, resources, etc.

• Dino Dai Zovi
• Vincenzo Iozzo, Zynamics
• Neil Archibald (nemo)
• Ruud van Baar, NFI
• Feico Dillema, NFI
• Raoul Bhoedjang, NFI