Subverting the FreeBSD ABI subsystem for phun and profits
The mechanism properties

- **Flexible**: enough polymorphic to be exploited in a lot of different ways
- **Simple to apply**: not passing through .text overwriting or /dev/kmem usage
- **Opaque**: very difficult to discover
- **Fast**: not eating too many system resources
ABI: Application Binary Interface

• It is from SCO Unix

• It lets a FreeBSD system run a lot of different binary types (Linux, SVR4, Solaris)

• It foresees a kernel support for anything needs to switch in the binary layout (i.e: syscalls handling)

• Issue: it loses in performance due to switching mechanisms
In sys/sys/sysent.h:

```c
struct sysentvec {
    int sv_size;    /* number of entries */
    struct sysent *sv_table;    /* syscall table */
    u_int sv_mask;    /* mask to index */
    int sv_errsize;  /* error table */
    int *sv_errtbl;
    int *sv_sigcode;
    void (*sv_prepsyscall)(struct trapframe *, int *,
                           u_int *, caddr_t *);    /* syscall parameters */
...
```
FreeBSD-i386: Making a syscall

In sys/i386/i386/exception.s:

```
SUPERALIGN_TEXT
IDTVEC(int0x80_syscall)                # builds a trapframe;
pushl  $2                               # using a trap gate
subl  $4,%esp
pushal
pushl  %ds
pushl  %es
pushl  %fs
SET_KERNEL_SREGS
FAKE_MCOUNT(TF_EIP(%esp))              # profiling stub
  call  syscall                        # syscall handling routine
MEEXITCOUNT
  jmp    doreti
```
FreeBSD-i386: Making a syscall

In sys/i386/i386/trap.c:

```c
void
syscall(struct trapframe frame)
{
    params = (caddr_t)frame.tf_esp + sizeof(int);  /* skipping ret value */
    code = frame.tf_eax;
    if (p->p_sysent->sv_prepsyscall) {       /* syscall parameters handling */
        (*p->p_sysent->sv_prepsyscall)(&frame, args, &code, &params);
    } else {
        if (code == SYS_syscall) {
            code = fuword(params);
            params += sizeof(int);
        } else if (code == SYS___syscall) {
            code = fuword(params);
            params += sizeof(quad_t);
        }
    }
    if (code >= p->p_sysent->sv_size)              /* getting the syscall table */
        callp = &p->p_sysent->sv_table[0];
    else
        callp = &p->p_sysent->sv_table[code];
    error = copyin(params, (caddr_t)args, (u_int)(narg * sizeof(int))); /* parameters */
    if (error == 0) {
        error = (*callp->sy_call)(td, args);   /* start syscall */
        ...
```
The idea

It is possible manipulating process behaviour working on syscall parameters gathering. In particular we can:

• **Choosing** a long living thread (possibly a simply controllable one)

• **Injecting** a preparing parameters handler which will shadow malicious code

• For further use, **forcing** malicious code to work with a pre-selected condition
rootkit_one: an implementation

rootkit_one is a simple root-suiding module for a specified shell. Basically:

• It *infects* “init” application and waits for an execve(“/usr/libexec/getty”, ...);

• The started malicious code scans all the processes in the kernel looking for “ABIrtk”.

• If “ABIrtk” binary is found, starting shell credentials *are updated* to root
ABI hijacking: considerations

• This technique flexibility ensures a good opacity to automatic tools

• In the case of root_one, search has linear complexity but it is called just very few times (fast)

• Rootkit triggering is very flexible and simple to do

• The rootkit is a KLD so it needs securelevel < 1 (sometimes it can be a problem)
Reference bibliography

• The design and implementation of the FreeBSD operating system

• The FreeBSD developers handbook

• The FreeBSD architecture handbook

• Various phrack magazine issues about kernel rootkits
Thanks

• Thanks to FreeBSD Foundation for sponsoring

• Thanks to the GUFI (FreeBSD Italian UserGroup), BSDCan and FreeBSD people for support