

Finding User/Kernel Pointer Bugs in FreeBSD

Sophie Engle, Sean Whalen*

March 20, 2009

Abstract

CQUAL is a type qualifier inference tool used to detect multiple classes of security vulnerabilities. This tool has been used to detect format string vulnerabilities, deadlock, and user/kernel pointer bugs in Linux. In this paper we discuss our extension of this work towards finding user/kernel pointer bugs in FreeBSD 5.3.

1 Introduction

An operating system kernel runs in a separate address space from user code and data. This protection mechanism aims to prevent user applications from crashing or manipulating the system. However, the kernel provides services to user applications and thus must interact with user space. This creates the potential for bypassing memory protection mechanisms.

The C language used to develop many kernels cannot distinguish between user space and kernel space pointers due to weak typing. The CQUAL tool extends the typing capabilities of C through programmer-annotated type qualifiers. These new qualifiers can be used to detect a variety of flaws including user/kernel pointer misuse. This paper discusses work applying CQUAL to the FreeBSD 5.3 kernel and our results, which should easily extend to all BSD systems.

2 Background

We first provide a brief overview of system calls, user/kernel pointer bugs, and the CQUAL type qualifier inference tool. Familiar readers may want to skip to Section 3.

2.1 System Calls in User Space

A kernel must be protected from user applications to provide safe, secure operation of the system. FreeBSD segments the virtual addresses space, giving the first 3 gigabytes to user and the last

*Department of Computer Science, University of California, Davis, USA, {engle, whalen}@cs.ucdavis.edu

gigabyte to kernel. Faults are generated when user applications directly attempt to read or write past this boundary.

However, user applications must cross this boundary to request services from the kernel. This is accomplished with special functions called system calls (syscalls). FreeBSD 5.3 has 296 system calls such as `exit`, `fork`, `read`, `write`, and `open`. Each syscall is assigned a unique number by the OS.

A C/C++ application invokes a syscall by calling a user space function in the C library, ex: `open(char* path, int flags, int mode)`. This stub pushes the syscall arguments on the stack and stores the syscall number in the `eax` register. To pass control to the kernel, the stub issues a trap by executing a supervisor-only instruction. The code below¹ shows this process for `open`:

```
kernel:
    int     80h           ; issue trap
    ret

open:
    push   dword mode
    push   dword flags
    push   dword path
    mov    eax, 5         ; open is syscall 5
    call   kernel
    add    esp, byte 12
    ret
```

The kernel now takes over, executing a trap handler (in `i386/i386/trap.c`) which recognizes `int 80h` as a syscall. The syscall's arguments are copied into kernel space, packed into a structure, and the syscall corresponding to the number in `eax` is executed. Packing the arguments is specific to FreeBSD.

2.2 System Calls in Kernel Space

Syscalls are specified by a number stored in `eax` before executing `int 80h`. FreeBSD declares syscall numbers in `sys/syscall.h`:

```
#define SYS_syscall 0
#define SYS_exit    1
#define SYS_fork    2
#define SYS_read    3
#define SYS_write   4
#define SYS_open    5
...
```

These numbers index into the syscall table, declared in `kern/init_sysent.c`:

¹<http://www.int80h.org/bsdasm/>

```

struct sysent sysent[] = {
    { SYF_MPSAFE | 0, (sy_call_t *)nosys },           /* 0 = syscall */
    { SYF_MPSAFE | AS(sys_exit_args), (sy_call_t *)sys_exit }, /* 1 = exit */
    { SYF_MPSAFE | 0, (sy_call_t *)fork },           /* 2 = fork */
    { SYF_MPSAFE | AS(read_args), (sy_call_t *)read }, /* 3 = read */
    { SYF_MPSAFE | AS(write_args), (sy_call_t *)write }, /* 4 = write */
    { SYF_MPSAFE | AS(open_args), (sy_call_t *)open }, /* 5 = open */
    ...
};

```

Both `sy_call_t` and `sysent` are defined in `sys/sysent.h`:

```

typedef int sy_call_t(struct thread *, void *);

struct sysent {
    int sy_narg;           /* number of arguments */
    sy_call_t *sy_call;   /* implementing function */
};

```

The `void*` parameter in `sy_call_t` is a pointer to a structure containing the syscall's arguments. Each syscall has an argument structure defined in `sys/sysproto.h`. Below is the structure for the `open` syscall, which includes macros for packing and alignment to the left and right of actual parameters:

```

struct open_args {
    char path_l_[PADL_(char *)]; char * path; char path_r_[PADR_(char *)];
    char flags_l_[PADL_(int)]; int flags; char flags_r_[PADR_(int)];
    char mode_l_[PADL_(int)]; int mode; char mode_r_[PADR_(int)];
};

```

In summary, each entry in the `sysent[]` table is a `sysent` structure containing the number of arguments and a pointer to the syscall's implementation. The implementation is passed a pointer to the calling thread and a structure containing user arguments. For example, `open`'s kernel space prototype becomes `int open(struct thread* td, struct open_args* uap)`.

2.3 User/Kernel Pointer Bugs

A program invoking a syscall passes arguments which exist in user space. The kernel can read from user space but not vice versa. A bug exists if a program can cause the kernel to dereference a user pointer, or if the kernel returns a kernel pointer to a program. The user could crash the system, corrupt kernel data, or escalate privileges.

Consider the following kernel code, which serves only to illustrate 2 user/kernel pointer bugs:

```

1 struct teststruct { char* buf; };
2 struct testsyscall_args { struct teststruct* ts; };
3
4 char* testsyscall(struct thread* td, testsyscall_args* args)

```

```

5 {
6     int len = strlen(args->ts->buf);
7     char* kbuf = (char*)malloc(10*sizeof(char));
8     return kbuf;
9 }

```

Error 1: (Line 8) A pointer to kernel space is returned. A user program accessing this pointer will segfault. Instead, kernel code should use the `copyout` function to copy intended data into user space.

Error 2: (Line 6) The kernel's `strlen` function is called with a pointer to a user structure. The kernel will panic when dereferencing `args->ts->buf` if `ts == NULL`. In a more complicated example, privilege escalation could occur if the user passes in a kernel pointer which the kernel writes to. Instead, kernel code should use the `copyin` function to validate and copy user data into kernel space.

These functions are defined in `sys/system.h`:

```

int copyout(const void* kaddr, void* uaddr, size_t len);
int copyin(const void* uaddr, void* kaddr, size_t len);

```

Caveat: `copyin` does not look for user pointers in a structure's fields. It will copy a structure's contents (fields) to kernel space, but a field's content may be a user address. This allows for more subtle bugs:

```

1 struct teststruct { char buf[10]; struct teststruct* next; };
2 struct testsyscall_args { struct teststruct* ts; };
3
4 void testsyscall(struct thread* td, testsyscall_args* args)
5 {
6     char kbuf[10] = "teststring";
7     struct teststruct kts;
8
9     copyin(args->ts, kts, sizeof(testsyscall_args));
10    memcmp(kts->next->buf, &kbuf, sizeof(kbuf));
11 }

```

The `args->ts` structure is first copied to kernel space (line 9). However, the content of the `kts->next` field is still a user address. If `kts->next == NULL`, dereferencing `kts->next->buf` will panic the kernel (line 10). Previous work in [1] found such user/kernel pointer bugs to be the most common.

This finding is essential: even with proper use of `copyin`, *user pointers may still be present in kernel space*.

2.4 Cqual

Variables in C are declared with a *type*, such as the familiar `int`, `char*`, or `float`. In addition, we can use *type qualifiers* such as `const` or `unsigned` to place restrictions on the type. The CQUAL² tool allows a programmer to define custom type qualifiers. Programmers first define the names and rules for the new qualifiers (or use those packaged with the tool), then annotate their source with these qualifiers. CQUAL is run on the annotated source and reports when qualifier rules are broken.

We are interested in the `$user` and `$kernel` type qualifiers developed in [1]. A pointer annotated with `$user` is assumed to come from user space and is untrusted. A pointer annotated with `$kernel` is assumed to come from kernel space and is trusted. Our primary concern is finding instances where a `$user` pointer is dereferenced or assigned to a `$kernel` pointer which is later dereferenced. Consider the following annotated kernel code:

```
1 char* $user ubuf = "user buffer";
2 char* $kernel kbuf = "kernel buffer";
3 kbuf = ubuf;
4 printf("%s", kbuf);
```

The user pointer `ubuf` is assigned to the kernel pointer `kbuf` and dereferenced during `printf`. CQUAL will statically detect this error, and is guaranteed sound (modulo some assumptions) for all classes of bugs being checked. False positives, however, are quite common and remain a significant obstacle for large projects. Many of these are due to the tool's broken flow sensitivity.

Annotations are minimized using type qualifier *inference*. By annotating function prototypes, variables passed as parameters to these functions have their type automatically inferred without explicit annotation. Consider another example:

```
1 int copyout(const void * $kernel, void * $user, size_t);
2
3 void testfunc(char* ubuf, size_t len)
4 {
5     char thirdchar;
6     char* kbuf = "kernel buffer";
7     copyout(kbuf, ubuf, len);
8     thirdchar = ubuf[3];
9 }
```

The second parameter to `copyout` is annotated as `$user`, so `ubuf` is inferred as `$user` on line 8. As a result, CQUAL will detect the dereferencing of a user pointer in the next statement. The inference process reduces manual annotations to a few key functions. In fact, we annotated only 22 functions for the entire FreeBSD kernel excluding syscalls.

More complicated annotations are necessary for finding user/kernel pointer bugs:

```
int copyin(const void * $user, void $user * $kernel, size_t);
$$a _op_deref($$a * $kernel);
```

²<http://cqual.sourceforge.net>

The `void $user * $kernel` parameter of `copyin` says the pointer is in kernel space, but its contents may point to user space. This corresponds to the structure fields bug discussed in Section 2.3, and thus is extremely important to understand.

The `_op_deref` entry allows annotation of the dereference operator. The annotation above says that for any pointer of type α , it may be dereferenced only if its qualifier is `$kernel`. The result of the dereference operator is also of type α . Without this annotation we would find very few bugs.

We now discuss our experimental setup and results. For more details on CQUAL, see [1].

3 Setup

Our initial environment was FreeBSD 5.3 running under the VMware³ 4.5.2 virtual machine for Windows XP. CQUAL 0.991 was installed and required several source changes to compile under FreeBSD. After discussion with developer Rob Johnson, we upgraded to CVS and applied additional source and configuration changes.

The main `cqual` executable runs on preprocessed C code generated by `gcc -E`. Without this flag, preprocessed code is piped from memory to `cc` and not written to disk. The `gcqual` script calls `.c` files and passes the resulting `.i` files to `cqual`. The `kqual` script passes several parameters helpful for kernel analysis to `gcqual`. The resulting control flow is `kqual` \rightarrow `gcqual` \rightarrow `cqual`.

3.1 Kernel Preprocessing

We could not integrate CQUAL into the FreeBSD kernel build process, so instead we explicitly use `gcc -E`. The build process then becomes:

```
cd /usr/src/sys/i386/config
cp GENERIC NEWCONFIG
config NEWCONFIG
cd ../compile/NEWCONFIG
make depend
make CC="gcc -save-temps"
```

This makes a copy of the `GENERIC` kernel config file, checks its syntax, changes to the new build directory, creates dependency lists, and compiles. Afterwards, the kernel can be audited with `kqual *.i 2>&1 | more`. This merges `stderr` with `stdout` (Bash shell syntax) and pipes the results to a pager for scrolling output.

CQUAL is memory-intensive for large programs such as a kernel, and running in a virtual environment exhausted available memory quickly. Because no object code is generated, preprocessed `.i` files can be transparently analyzed on any system where the tool will run. Since our virtual environment had only 512 MB of RAM, we decided to analyze the kernel's preprocessed source on a non-virtual Linux system with 1 GB of RAM.

³<http://www.vmware.com>

Unfortunately, memory was still a primary limitation. Previous work with Linux required 10 GB of RAM to effectively analyze the entire kernel. We hoped to find true positives in spite of our memory constraints by first analyzing individual files, then kernel subsystems. We wrote the `fqual` Python script to traverse the FreeBSD source tree and pass subsystem-related files to `kqual`. The script was executed separately for each subsystem in the tree.

3.2 Configuration Files

Extended type qualifiers and their partial orderings are defined in `cqual/share/cqual/lattice`. An example lattice is included with `CQUAL` which we stripped of all types except `$user` and `$kernel` to reduce memory usage:

```
partial order [casts-preserve] {
    $kernel [level=value, color="pam-color-4", sign=eq, fieldptrflow=all]
    $user [level=value, color="pam-color-6", sign=eq, ptrflow=down,
          fieldflow=down, fieldptrflow=all]
}
```

Annotation files are placed in `cqual/share/cqual` and must contain valid C code. We used `syscalls.cq` for syscalls, but later removed the file due to memory overhead. Initial results showed few errors related to syscalls, which may have resulted from lack of inter-file analysis on the entire kernel. We used `noderef.cq` for kernel functions and operators:

```
struct malloc_type;

typedef unsigned int __uint32_t;
typedef __uint32_t u_int32_t;
typedef u_int32_t size_t;

char * $kernel memscan(void * $kernel, unsigned char, int);
char * $kernel strcat(char * $kernel, const char * $kernel);
char * $kernel strcpy(char * $kernel, const char * $kernel);
char * $kernel strdup(const char * $kernel, struct malloc_type * $kernel);
char * $kernel strncpy(char * $kernel, const char * $kernel, size_t);
char * $kernel strstr(char * $kernel, char * $kernel);
char * $kernel strncat(char * $kernel, const char * $kernel, size_t);

int copyin(const void * $user, void $user * $kernel, size_t);
int copyinstr(const void * $user, void * $kernel, size_t, size_t *);
int copyout(const void * $kernel, void * $user, size_t);
int copystr(const void * $kernel, void * $kernel, size_t, size_t *);
int memcmp(const void * $kernel, const void * $kernel, size_t);
int strcmp(const char * $kernel, const char * $kernel);
int strncmp(const char * $kernel, const char * $kernel, size_t);
int strvalid(const char * $kernel, size_t);

size_t strlcat(char * $kernel, const char * $kernel, size_t);
size_t strlcpy(char * $kernel, const char * $kernel, size_t);
size_t strlen(const char * $kernel);
```

```

void * $kernel memcpy(void * $kernel, const void * $kernel, size_t);
void * $kernel memset(void * $kernel, int, size_t);

void free($$a * $kernel, struct malloc_type * $kernel);

$$a * $kernel malloc(unsigned long, struct malloc_type * $kernel, int);
$$a _op_deref($$a * $kernel x);

```

Interestingly, reducing this file to only `copyin`, `copyout`, and `_op_deref` found roughly the same amount of errors when run on individual files. This shows the importance of inter-file analysis in leveraging the tool.

4 Results

Several individual files caused `kqual` to fail, confirmed under both Linux and FreeBSD. These files were renamed `*.err` so subsystem analysis could proceed (Figure 1). Most failures were due to array size assertions or parsing.

preprocessed files causing failure		
cam/cam_xpt.i	i386/i386/dump_machdep.ifs/ffs/ffs_balloc.i	
dev/advansys/adwlib.i	i386/isa/atpic.i	ufs/ffs/ffs_inode.i
dev/cs/if_cs.i	kern/kern_descrip.i	ufs/ffs/ffs_snapshot.i
dev/fdc/fdc.i	kern/kern_proc.i	ufs/ffs/ffs_softdep.i
dev/hptmv/entry.i	netinet/tcp_sack.i	ufs/ffs/ffs_subr.i
dev/usb/umass.i	nfs4client/nfs4_vnops.i	ufs/ffs/ffs_vfsops.i
geom/geom_gpt.i	ufs/ffs/ffs_alloc.i	ufs/ffs/ffs_vnops.i

Figure 1: Individual files causing `kqual` to fail.

Some subsystems required too much memory, resulting in `kqual` failure (Figure 2). We found a maximum number of analyzable files for each subsystem. We then ran `kqual` with a random permutation of the maximum number of files. This was repeated 10 times for each subsystem, and we report the average below.

memory intensive subsystems			
subsystem	total files	file max	warnings
dev/mii	26	24	0
kern	50	40	11
pci (with if_ti.i)	22	15	22
pci (w/o if_ti.i)	21	17	0

Figure 2: Subsystems causing `kqual` failure by exhausting available memory. Columns are total files in the subsystem, maximum analyzed before failure, and average number of resulting warnings.

preprocessed warning summary					
subsystem	file name	w	f	i	u
dev/aac/	aac.c	3	2	1	0
dev/an/	if_an.c	1	0	0	1
dev/asr/	asr.c	19	0	0	19
dev/mlx/	mlx.c	2	0	0	2
fs/procfs/	procfs_status.c	1	0	1	0
geom/	geom_ctl.c	3	3	0	0
i386/i386	sys_machdep.c	2	0	0	2
i386/i386	trap.c	1	1	0	0
kern/	kern_event.c	1	1	0	0
kern/	kern_ktrace.c	1	1	0	0
kern/	kern_linker.c	1	1	0	0
kern/	kern_subr.c	2	1	1	0
kern/	kern_sysctl.c	7	4	0	3
kern/	sys_process.c	2	2	0	0
kern/	sysv_sem.c	1	1	0	0
kern/	uipc_syscalls.c	2	2	0	0
kern/	vfs_lookup.c	1	1	0	0
kern/	vfs_mount.c	1	1	0	0
kern/	vfs_syscalls.c	2	2	0	0
nfsclient/	nfs_vfsops.c	1	1	0	0
pci/	if_ti.c	11	0	0	11
sys/	mbuf.h	1	0	0	1
vm/	vm_object.h	1	0	0	1
totals:		67	24	3	40

subsystem warning summary					
subsystem	c	w	f	i	u
cam	13	7	0	0	7
contrib	3	2	0	1	1
dev/aac	5	1	0	0	1
dev/ata	13	9	0	0	9
geom	13	7	0	2	5
i386	40	23	4	0	19
net	23	7	7	0	0
nfsclient	9	1	1	0	0
totals:	119	57	12	3	42

Figure 3: Number of kqual warnings for individual files and subsystems excluding dev/mii, kern, and pci. Columns are (c)hecked files, (w)arnings reported, (f)alse positives, (i)nteresting warnings, and (u)nanalyzed warnings.

Figure 3 summarizes kqual warnings for 753 individual preprocessed files and 8 subsystems after pruning shared warnings. In hindsight, individual file analysis was unnecessary since shared errors will also be detected during subsystem analysis. Unlisted subsystems either had no preprocessed files, produced no new warnings, or exhausted memory. Figure 4 summarizes common kqual warning messages (excluding traces).

common kqual warnings		
#	%	warning message
33	27%	type of argument doesn't match type of formal
33	27%	incompatible types in assignment
19	15%	operands incompatible with operator definition
16	13%	incompatible qualifiers at cast
14	11%	incompatible operands of + or -
08	06%	variable treated as \$kernel and \$user
01	01%	return type incompatible with function type

Figure 4: Common kqual warnings for individual files and subsystems excluding dev/mii, kern, and pci.

We are not expert FreeBSD kernel hackers, and thus labeled false positives conservatively. Many warnings were difficult to classify due to use of functions like `copyin` with two `$kernel` parameters. It was also difficult to identify which function parameters came from user space based on the prototype.

If a warning was too long or undecipherable it was labeled “unanalyzed”. Potentially valid bugs were marked “interesting” and will be presented to the FreeBSD community for validation.

4.1 Classification

Here we examine two warnings and demonstrate our classification process. First, consider the warning generated for `fs/procfs/procfs_status.c`:

```

../../../../fs/procfs/procfs_status.c:204 Operands incompatible with operator definition
pstr.ps_argvstr: $kernel $user
/usr/local/cqual/share/cqual/noderef.cq:34      $kernel == _op_deref_arg1@34@204
../../../../fs/procfs/procfs_status.c:204      == pstr.ps_argvstr
../../../../fs/procfs/procfs_status.c:176      == pstr
../../../../fs/procfs/procfs_status.c:199      == *copyin_arg2@199
../../../../fs/procfs/procfs_status.c:199      == *copyin_arg2
/usr/local/cqual/share/cqual/noderef.cq:15      == $user

```

Let’s look at this file and trace through the warning:

```

173 int
174 procfs_doproccmdline(PFS_FILL_ARGS)
175 {
* 176     struct ps_strings pstr;

196     if (p != td->td_proc) {
197         sbuf_printf(sb, "%. *s", MAXCOMLEN, p->p_comm);
198     } else {
* 199         error = copyin((void *)p->p_sysent->sv_psstrings, &pstr,
200             sizeof(pstr));
201         if (error)
202             return (error);
203         for (i = 0; i < pstr.ps_nargvstr; i++) {
* 204             sbuf_copyin(sb, pstr.ps_argvstr[i], 0);
205             sbuf_printf(sb, "%c", '\0');
206         }
207     }

210 }

```

On line 176, `pstr` is a locally declared `$kernel` variable. On line 199 it is the second parameter to `copyin` and thus has contents under user control. On line 204 those contents are dereferenced by `pstr.ps_argvstr[i]`. Unless the first argument of `copyin` on line 199 has contents in kernel space, this is a valid user/kernel bug.

The next warning comes from `kern/vfs_lookup.c`:

```
../../../../kern/vfs_lookup.c:125 type of actual argument 1 doesn't match type of formal
ndp->ni_dirp: $kernel $user const
/usr/local/cqual/share/cqual/noderef.cq:18      $kernel == copystr_arg1
          ../../../../kern/vfs_lookup.c:125      == copystr_arg1@125
          ../../../../kern/vfs_lookup.c:125      == ndp->ni_dirp
          ../../../../kern/vfs_lookup.c:128      == copyinstr_arg1@128
          ../../../../kern/vfs_lookup.c:128      == copyinstr_arg1
/usr/local/cqual/share/cqual/noderef.cq:16      == $user
```

Relevant lines from `noderef.cq`:

```
16 int copyinstr(const void * $user, void * $kernel, size_t, size_t *);
18 int copystr(const void * $kernel, void * $kernel, size_t, size_t *);
```

The offending code:

```
124 if (ndp->ni_segflg == UIO_SYSSPACE)
* 125     error = copystr(ndp->ni_dirp, cnp->cn_pnbuf,
126                     MAXPATHLEN, (size_t *)&ndp->ni_pathlen);
127 else
* 128     error = copyinstr(ndp->ni_dirp, cnp->cn_pnbuf,
129                       MAXPATHLEN, (size_t *)&ndp->ni_pathlen);
```

Here we see `ndp->ni_dirp` is used as both `$user` and `$kernel`, but never simultaneously. This is a false positive resulting from broken flow sensitivity in CQUAL.

5 Conclusion

User/kernel pointer bugs can result in serious security flaws and are often subtle, spanning multiple files and dozens of assignments and casts. An easy to use, sound tool like CQUAL helps programmers find such bugs which are too difficult to detect manually. High false positive rates and memory requirements remain practical roadblocks to leveraging this tool for large software projects.

Our work has three main contributions towards detecting user/kernel pointer bugs:

- The migration of CQUAL to FreeBSD and integration with the kernel build process. User and kernel code can now be natively analyzed on most BSD systems. By using a virtual machine, code can be preprocessed, extracted, and analyzed on the host OS of the user's choice.
- The annotation of BSD syscalls and kernel functions.
- The identification of at least 5 potential user/kernel pointer bugs in FreeBSD 5.3.

Some suggestions for future work:

- Inter-file analysis of the Linux kernel in [1] required 10 gigabytes of RAM. Using similar amounts of memory, a kernel-wide inter-file analysis of FreeBSD using syscall annotations could greatly increase true positives.
- Analysis of other BSD kernels. Using only `copyin`, `copyout`, and dereference annotations should give results for OpenBSD, NetBSD, and Darwin. Other annotations may need small changes due to OS variations.
- Confirmation of potential bugs with the BSD community.

This brings us to a major consideration: CQUAL is best used by experts of the code. Due to our unfamiliarity with the kernel, we classified long or confusing traces as “unknown” and likely missed some true positives. This was necessary due to time constraints, given the large number of complex traces and false positives. A re-implementation of flow sensitivity would simplify analysis by reducing the number of both simple and complex false positives.

Thanks to Rob Johnson for his many helpful comments, and the rest of the CQUAL team.

References

- [1] R. Johnson and D. Wagner, “Finding User/Kernel Pointer Bugs With Type Inference”, *13th USENIX Security Symposium*, 2004.