```
struct jail {
    uint32_t                              version;
    __uaddr_str char * __capability path;
    __uaddr_str char * __capability hostname;
    __uaddr_str char * __capability jailname;
```

The copyinout Framework

ABI-independent, type-aware, capability-aware, copyin and copyout API in FreeBSD and CheriBSD

BY KONRAD WITASZCZYK

Memory copying between user and kernel address spaces is a crucial operation performed as part of system calls. It is used to copy system call arguments as well as a system call result. The current copy function prototypes are type-agnostic and copy a number of bytes from an arbitrary buffer. When a kernel copies its memory to the user space, it must make sure it does not leak any kernel data that might include secrets. This article describes the limitations of the current copy functions in FreeBSD and CheriBSD [1] and proposes a framework that could improve the security and code quality of system call handlers.

The described copyinout framework was implemented as part of the MSc thesis entitled "Capability-aware memory copying between address spaces" [2] under supervision from Ken Friis Larsen, University of Copenhagen, and David Chisnall, Microsoft Research Ltd. The original idea of the type-aware copyin and copyout API was proposed by David Chisnall.

Memory Copy Functions in FreeBSD

FreeBSD includes two main functions that copy memory between address spaces: `copyin()` and `copyout()` (see Listing 1). Both functions take three arguments: a source address, a destination address, and a number of bytes to be copied. `copyin()` copies `len` bytes from the user-space address `udaddr` to the kernel-space address `kaddr`. `copyout()` works in the opposite direction and copies `len` bytes from the kernel-space address `kaddr` to the user-space address `udaddr`. The functions return `0` on success and `EFAULT` if an invalid address was passed.

```
int copyin(const void * __restrict udaddr, void * _Nonnull __restrict kaddr, size_t len);
int copyout(const void * _Nonnull __restrict kaddr, void * __restrict udaddr, size_t len);
```

Listing 1. `copyin()` and `copyout()` function prototypes in FreeBSD 13.0-RELEASE.

Directly from the function prototypes, we can identify one potential security issue. The copy functions operate on arbitrary buffers. In case a buffer contains a structure object with padding between structure fields, the padding is also copied. A padding leak with sensitive information is commonly known as a kernel memory disclosure or a kernel memory leak. Such bugs can result in escalated privileges. They are not specific to FreeBSD and can be found in numerous operating systems [3] [4] [5] as well as they have been a subject of extensive research of detection [6] and mitigation [7] [8] [9] techniques.

The copy functions are used by system call handlers to copy system call arguments from the user space to the kernel space and copy system call results from the kernel space to the user space. Since the system calls are very frequently executed, kernel developers must provide copy function implementations with the lowest possible overhead. This can be achieved by providing machine-dependent implementations in an assembly language. Depending on a CPU architecture, the copy functions can also make use of security features if a CPU model provides them. For example, implementations for amd64 (see `amd64/amd64/support.S`) support Supervisor Mode Access Prevention (SMAP).

ABI Support

FreeBSD includes support for multiple ABIs. In particular:

- Native ABI for programs compiled for the same target as the kernel;
- 32-bit ABI for a 32-bit version of an architecture for which the kernel was compiled;
- Linux ABI for Linux user-space programs.

The ABIs are implemented as compatibility layers. Each compatibility layer provides its system call handlers that implement additional logic required to be executed before or after the kernel enters or returns from kernel routines that operate on native ABI objects. This includes copying and translating objects between address spaces, e.g. for the 32-bit ABI, a pointer in a system call argument or a system call result must be translated from or to a 32-bit pointer.

System Call Handlers

The kernel calls a specific system call handler with copied in system call arguments each time a user-space program makes a system call. For each supported ABI, the kernel keeps a `sysentvec` structure object (see Listing 2) describing ABI-specific properties and functions to be used by the kernel when a program is being executed. The structure includes the `sv_table` array with system call handler function pointers at stored positions specified by their corresponding system call numbers, and the `sv_fetch_syscall_args` function pointer to an architecture-specific function that copies in system call arguments.

As an example, let's consider the `jail(2)` system call. This system call has one argument: a pointer to a `jail` structure object (see Listing 3). The `jail` structure includes parameters describing the prison (see Listing 4). Once a user-space program performs the `jail` system call and enters the privileged mode, the kernel calls the `jail` system call handler — the `sys_jail()` function (see Listing 5) with already copied-in jail system call arguments — a `jail_args` structure object. The system call handler copies in a `jail` structure and calls the `kern_jail` kernel routine that implements `jail` system call logic.

```
struct sysentvec {
   (...)
   struct sysent *sv_table;
   (...)
   int            (*sv_fetch_syscall_args)(struct thread *);
   (...)
}
```

Listing 2. `sysentvec` structure describing ABI-specific properties and functions.

```
struct jail_args {
    char jail_l_[PADL_(struct jail *)];
    struct jail * jail;
    char jail_r_[PADR_(struct jail *)];
};
```

Listing 3. `jail` system call arguments structure for the native ABI.

```
struct jail {
    uint32_t                version;
    char                   *path;
    char                   *hostname;
    char                   *jailname;
    uint32_t                ip4s;
    uint32_t                ip6s;
    struct in_addr         *ip4;
    struct in6_addr        *ip6;
};
```

Listing 4. `jail` structure for the native ABI.

```
int
sys_jail(struct thread *td, struct jail_args *uap)
{
    (...)
    int error;
    struct jail j;

    (...)
           error = copyin(uap->jail, &j, sizeof(struct jail));
           if (error)
                return (error);
    (...)
    return (kern_jail(td, &j));
}
```

Listing 5. `jail` system call handler for the native ABI.

Compatibility Layers

Compatibility layers provide system call implementations for non-native ABIs. In particular, the 32-bit ABI is implemented as the freebsd32 compatibility layer. Let's consider the same `jail` system call for the 32-bit ABI. The 32-bit version of the `jail` arguments structure is called `freebsd32_jail_args` (see Listing 6) and includes a pointer to an object of the 32-bit version of the `jail` structure called `jail32` (see Listing 7). The only differences between the `jail` and `jail32` structures are architecture-independent field types. Each pointer is replaced with a 32-bit unsigned integer. Since pointers are 32-bit unsigned integers in 32-bit architectures, this change guarantees that the `jail32` structure compiled for a 64-bit kernel has the same layout as the jail structure compiled for a 32-bit kernel.

The `jail` system call handler for the 32-bit ABI is implemented as the `freebsd32_jail` function (see Listing 8). The function copies in a 32-bit jail object, translates each field for its native ABI version using macros (see Listing 9) and calls the same **kern_jail** kernel routine as in the native ABI case. This means that the only difference between the native ABI and the 32-bit ABI in the **jail** system call handler is translating a user-space **jail** object to its native ABI version that can be used by the kernel.

```
struct freebsd32_jail_args {
    char jail_l_[PADL_(struct jail32 *)];
    struct jail32 * jail;
    char jail_r_[PADR_(struct jail32 *)];
};
```

Listing 6. `jail` system call arguments structure for the 32-bit ABI.

```
struct jail32 {
    uint32_t        version;
    uint32_t        path;
    uint32_t        hostname;
    uint32_t        jailname;
    uint32_t        ip4s;
    uint32_t        ip6s;
    uint32_t        ip4;
    uint32_t        ip6;
};
```

Listing 7. `jail` structure for the 32-bit ABI.

```
int
freebsd32_jail(struct thread *td, struct freebsd32_jail_args *uap)
{
     (...)
     int error;
     struct jail j;

     (...)
         struct jail32 j32;

         error = copyin(uap->jail, &j32, sizeof(struct jail32));
         if (error)
               return (error);
         CP(j32, j, version);
         PTRIN_CP(j32, j, path);
         PTRIN_CP(j32, j, hostname);
         PTRIN_CP(j32, j, jailname);
         CP(j32, j, ip4s);
         CP(j32, j, ip6s);
         PTRIN_CP(j32, j, ip4);
         PTRIN_CP(j32, j, ip6);
     (...)
     return (kern_jail(td, &j));
}
```

Listing 8. `jail` system call handler for the 32-bit ABI.

```
#define CP(src, dst, fld) do {             \
     (dst).fld = (src).fld;                \
} while (0)

#define PTRIN(v) (void *)(uintptr_t)(v)
#define PTRIN_CP(src, dst, fld) do {       \
     (dst).fld = PTRIN((src).fld);         \
} while (0)
```

Listing 9. Helper macros used by the `jail` system call handler for the 32-bit ABI.

The copyinout API

Kernel memory disclosure and code duplication issues described in the previous sections are implications of type-unawareness of the copy functions. If `copyin()` and `copyout()` functions were aware of a structure of an underlying object stored in a copied buffer, they could copy fields of the objects and translate them if a user process ABI differs from the kernel ABI.

To eliminate these problems, we introduce the `copyinout API`. For each type `foo` that is copied between the user space and the kernel space, we introduce type-aware copy function variants (see Listing 10) that copy each field of the type `foo` and translate them from a source ABI to a destination ABI, e.g. a 32-bit pointer is translated to a 64-bit pointer for a 32-bit process on a 64-bit architecture. Additionally, the copy functions can perform additional operations depending on a CPU model, e.g. a kernel compiled for a CHERI CPU [11] can create CHERI capabilities [12] to set bounds or permissions for a field. In contrast to the original copy functions, the type-aware copy functions take only two arguments: a source address and a destination address. `copyin_foo()` copies an object stored at the uaddr user-space address to an object stored at the kaddr kernel-space address. `copyout_foo()` works in the opposite and copies an object stored at the `kaddr` kernel-space address to an object stored at the `uaddr` user-space address. For example, the `jail` structure described in the previous sections could be copied in using the following function call:

```
copyin_jail(uap->jail, &j);
```

```
int copyin_foo(const void *uaddr, struct foo *kaddr);
int copyout_foo(const struct foo *kaddr, const void *uaddr);
```

Listing 10. `copyin()` and `copyout()` function variants for the type `foo`.

The copyinout Framework

Implementations of the type-aware copy functions are independent of the `copyinout API` itself. In order to provide the implementations, we introduce the `copyinout framework` that consists of:

- Type annotations describing what copy functions should be generated for a type;
- A table of copy function pointers for each ABI;
- A kernel interface to dynamically register copy functions and call them;
- A code-generating tool that generates copy functions based on the type annotations.

The kernel defines type annotations that indicate what copy function should be generated:

- `__copyin` to generate `copyin()`;
- `__copyout` to generate `copyout()`;
- `__copyinout` to generate both `copyin()` and `copyout()`.

Additionally, the kernel defines field annotations that describe what value is stored in a field:

- `__uaddr_array(bar)` for a field that stores a pointer to an array with a number of elements stored in the field `bar`;
- `__uaddr_bounded(bar)` for a field that stores a pointer to a buffer with a number of bytes stored in the field `bar`;
- `__uaddr_code` for a field that stores a code pointer;
- `__uaddr_object` for a field that stores a pointer to an object;
- `__uaddr_unbounded` for a field that stores a pointer to a buffer with an unknown number of bytes;
- `__uaddr_str` for a field that stores a pointer to a string and hence its bounds can be computed with `strlen()`.

The field annotations can be used to generate copy functions that make use of security-related mechanisms, e.g., construct CHERI capabilities. Having the `copyinout framework` integrated, a kernel developer who wants to generate new copy functions for the type `foo` defined in Listing 11 must only add appropriate annotations and run the code-generating tool. In this case, the generated copy functions copy the field `len` and translate the pointer stored in the pointer `array`. Additionally, in CheriBSD, they construct the bounded capability array with bounds set to a value stored in the field `len` as part of the field translation.

```
struct foo {                              struct foo {
    size_t len;                               size_t len;
    int *array;                               __uaddr_array(len) int * __capability array;
};                                        } __copyinout;
```

Listing 11. Type `foo` before and after `copyinout` changes.

In order to provide separate copy functions for different ABIs, the copyinout framework implements the `copyinout table` with type-specific `copyin()` and `copyout()` function pointers (see Listing 12) as part of the `sysentvec` structure (see Listing 13 as compared to Listing 2). Each ABI allocates its own `copyinout` table that is dynamically filled with entries using the SYSINIT framework through the Linker Set technique [10]. The `SYSINIT()` and `SYSUNINIT()` macro calls for the ABIs are generated by the code generating tool alongside generated copy functions. This allows generating copy functions that are used within a kernel module and should be registered and unregistered when the module is loaded and unloaded. With the `copyinout table`, the `copyinout API` can be defined as in-kernel macros that cast pointers and call functions from a `copyinout table` corresponding to an ABI of a currently running thread (see Listing 14). A `copyinout` table entry index for a specific function is a global variable initialized as part of the `SYSINIT()` call. For example, the type `foo` with its generated functions has an associated variable `copyinout_foo_idx` that can be used indirectly by the `COPYIN_CALL()` and `COPYOUT_CALL()` macros to determine function addresses and make function calls (see Listing 15).

```
typedef int copyin_t(const void * __capability uaddr, void * __capability kaddr);
typedef int copyout_t(const void * __capability kaddr, void * __capability uaddr);

struct copyinout {
     copyin_t       *ce_copyin;
     copyout_t      *ce_copyout;
};
```

Listing 12. `copyinout structure` with type-specific copy function pointers.

```
struct sysentvec {
    (...)
    struct sysent *sv_table;
    (...)
    int            (*sv_fetch_syscall_args)(struct thread *);
    (...)
    const struct copyinout *sv_copyinout;
}
```

Listing 13. `sysentvec structure` with the `copyinout` table.

```
#define THREAD_COPYINOUT(thread, type)                               \
     (thread)->td_proc->p_sysent->sv_copyinout[copyinout_##type##_idx]

#define COPYIN_FUN(type)                                             \
     THREAD_COPYINOUT(curthread, type).ce_copyin
#define COPYIN_CALL(type, uaddr, kaddr)                              \
     ((int (*)(const void * __capability,                            \
         struct type * __capability))COPYIN_FUN(type))               \
         (uaddr, kaddr)

#define COPYOUT_FUN(type)                                            \
     THREAD_COPYINOUT(curthread, type).ce_copyout
#define COPYOUT_CALL(type, kaddr, uaddr)                             \
     ((int (*)(const struct type * __capability,                     \
         void * __capability))COPYOUT_FUN(type))                     \
         (kaddr, uaddr)
```

Listing 14. In-kernel macros for `copyinout` API calls.

```
#define copyin_foo(uaddr, kaddr) \
     COPYIN_CALL(foo, uaddr, kaddr)
#define copyout_foo(kaddr, uaddr) \
     COPYOUT_CALL(foo, kaddr, uaddr)
```

Listing 15. `copyinout API` function call macros for the type `foo`.

The main part of the `copyinout framework` is the code-generating tool written in C++ that uses the `libclang` library for code analysis. It traverses input Clang AST trees of header files that include all type definitions for which copy functions should be generated and prints prototypes, definitions or implementations of the copy functions (see Listing 16). The AST trees can be generated with the following command:

```
clang -Xclang -ast-dump -fsyntax-only -c header-file
```

The `copyinout framework` includes a helper script that for an input base source tree generates AST trees, runs the `copyinout tool` and places generated functions in the base source tree. This script could be added to the FreeBSD build system to automate code generation.

Currently, the function implementations can be generated in the C language for any architecture `(generic)` or in the assembly language for MIPS and CHERI-MIPS architectures. For example, Listing 17 presents a `copyin()` function in the assembly language generated by the `copyinout tool` for the type `foo` (see Listing 11) and the native ABI (a hybrid CheriBSD kernel).

```
$ ./copyinout
usage: copyinout prototypes kernel-space-ast
       copyinout definitions native|freebsd32|cheri kernel-space-ast
       copyinout implementations native|freebsd32|cheri generic|mips user-space-ast
kernel-space-ast
```

Listing 16. `copyinout` tool usage.

```
struct foo {                                    | LEAF(native_copyin_foo)
                                                |     cgetbase t0 , $c3
                                                |     blt t0, zero, _C_LABEL(copyerr)
                                                |     nop
                                                |     GET_CPU_PCPU(v1)
                                                |     PTR_L t0, PC_CURPCB(v1)
                                                |     PTR_LA t1 , copyerr
                                                |
  size_t len;                                   |     PTR_S t1, U_PCB_ONFAULT(t0)
                                                |     cld v0, zero, 0($c3)
                                                |     PTR_S zero, U_PCB_ONFAULT(t0)
                                                |     csd v0, zero, 0($c4)
                                                |
  __uaddr_array(len) int * __capability array; |     PTR_S t1, U_PCB_ONFAULT(t0)
                                                |     cld v0, zero, 8($c3)
                                                |     cfromptr $c5 , $ddc , v0
                                                |     cld v0, zero, 0($c3)
                                                |     li a0, 96
                                                |     multu a0, v0
                                                |     mflo v0
                                                |     csetbounds $c5 , $c5 , v0
                                                |     PTR_S zero, U_PCB_ONFAULT(t0)
                                                |     csc $c5, zero, 16($c4)
                                                |
                                                |     j ra
                                                |     move v0, zero
} __copyinout;                                  | END(native_copyin_foo)
```

Listing 17. `copyin()` function generated for the type `foo` and the native ABI (a hybrid CheriBSD kernel).

Memory Copying with the copyinout API

Having the `copyinout framework`, we can simplify system call structures and handlers. For the previously discussed `jail` system call, we can replace the `jail` (see Listing 4) and `jail32` (see Listing 7) structures with a single one (see Listing 18). The `jail` system call for the native ABI (see Listing 5) can be modified to use the type-aware `copyin()` function variant `copyin_jail()` (see Listing 19). Since the `copyin_jail()` function is also ABI-independent and can automatically translate a 32-bit ABI object to its native ABI version, we can also modify the `jail` system call handler for the 32-bit ABI (see Listing 8) to simply call the `sys_jail()` function (see Listing 20).

```
struct jail {
    uint32_t                        version;
    __uaddr_str char * __capability path;
    __uaddr_str char * __capability hostname;
    __uaddr_str char * __capability jailname;
    uint32_t                        ip4s;
    uint32_t                        ip6s;
    __uaddr_array(ip4s) struct in_addr * __capability ip4;
    __uaddr_array(ip6s) struct in6_addr * __capability ip6;
} __copyinout;
```

Listing 18. `jail` structure with `copyinout` changes for all ABIs.

```
int
sys_jail(struct thread *td, struct jail_args *uap)
{
    (...)
    int error;
    struct jail j;

    (...)
            error = copyin_jail(uap->jail, &j);
            if (error)
                  return (error);
    (...)
    return (kern_jail(td, &j));
}
```

Listing 19. `jail` system call handler with `copyinout` changes for the native ABI.

```
int
freebsd32_jail(struct thread *td, struct freebsd32_jail_args *uap)
{
    struct jail_args args;

    args.jailp = uap->jailp;

    return (sys_jail(td, &args));
}
```

Listing 20. `jail` system call handler with `copyinout` changes for the 32-bit ABI.

Conclusion

The initial implementation of the `copyinout framework` shows that copy-function generation can improve code quality in system call handlers and eliminate potential kernel memory leaks in paddings. However, the code-generating tool must be improved to support more complex data types and generate assembly copy function implementations for all supported platforms. While work on the framework has been on hold for a while, we hope to resume it and implement these improvements soon.

Future Work

Besides the basic functionality for the `copyinout framework`, there are several ideas on how it can be improved or applied in the future:

- **Assembly copy function implementation optimizations;**
  The current assembly implementations do not use any optimization techniques to minimize the number of registers or cycles used during copying. For example, two half-word adjacent fields could be loaded as one word with one load instruction and one register instead of two load instructions.
- **System call handlers reductions;**
  Compatibility layers implement many system call handlers that only translate objects between ABIs and not introduce any additional logic to their native ABI counterparts. Having the `copyinout API` applied to them, the system call handlers seem to be redundant. For example, the `jail` system call handler with copyinout changes for the 32-bit ABI (see Listing 20) only copies a pointer to a `jail` object from a `jail` system call arguments structure and calls the `jail` system call handler for the native ABI. It would be interesting to apply the `copyinout framework` to system call argument structures as well and remove such system call handlers from the compatibility layers.
- **Cross-platform compatibility layers for emulators.**
  User-mode emulation in QEMU allows running programs for different architectures than a host architecture without full system emulation. Each time a system call is encountered, QEMU translates system call arguments from an emulated ABI version to a host user-space ABI version, performs a system call on a host and translates a result to its emulated ABI version. We could investigate if it is possible to implement a compatibility layer that includes system call handlers for an emulated platform and using the `copyinout framework` can copy and translate system call objects directly from or to an emulated user space.

References

[1] CTSRD. CheriBSD. FreeBSD adapted for CHERI-MIPS, CHERI-RISC-V, and Arm Morello. https://www.cl.cam.ac.uk/research/security/ctsrd/cheri/cheribsd.html.

[2] Konrad Witaszczyk. Capability-aware memory copying between address spaces. University of Copenhagen, 2019.

[3] The FreeBSD Project. FreeBSD-EN-18:12.mem. https://www.freebsd.org/security/advisories/FreeBSD-EN-18:12.mem.asc.

[4] The MITRE Corporation. CVE-2017-16994. https://cve.mitre.org/cgi-bin/cvename.cgi?name =CVE-2017-16994.

[5] The MITRE Corporation. CVE-2010-4082. https://cve.mitre.org/cgi-bin/cvename.cgi?name =CVE-2010-4082.

[6] Mateusz Jurczyk. Detecting Kernel Memory Disclosure with x86 Emulation and Taint Tracking. https://googleprojectzero.blogspot.com/2018/06/detecting-kernel-memory-disclosure.html.

[7] Alexander Popov. Introduce the STACKLEAK feature and a test for it. https://lwn.net/Articles/735584/.

[8] Kees Cook. mm: Hardened usercopy. https://lwn.net/Articles/693745/.

[9] Thomas Barabosch, Maxime Villard. KLEAK: Practical Kernel Memory Disclosure Detection. https://www.netbsd.org/gallery/presentations/maxv/kleak.pdf.

[10] The FreeBSD Project. FreeBSD Architecture Handbook, Chapter 5. The SYSINIT Frame-

work. https://docs.freebsd.org/en/books/arch-handbook/sysinit/.

[11] Robert N. M. Watson, et al. Capability Hardware Enhanced RISC Instructions: CHERI Instruction-Set Architecture (Version 8). Technical Report UCAM-CL-TR-951, University of Cambridge, Computer Laboratory, 2020.

[12] Robert N.M. Watson, et al. An Introduction to CHERI. Technical Report UCAM-CL- TR-941, University of Cambridge, Computer Laboratory, 2019.

KONRAD WITASZCZYK is a Research Associate at the University of Cambridge working on the CHERI project. He graduated with a BSc degree in Theoretical Computer Science from the Jagiellonian University, an MSc degree in Computer Science from the University of Copenhagen and spent almost 7 years at Fudo Security working with FreeBSD and its security-related technologies. Currently, Konrad lives in Warsaw, Poland.