Vita sceNetIoctl use-after-free

Aug 30, 2016

This describes the first public PS Vita kernel exploit. The vulnerability is a use-after-free in the SceNetPs module, combined with an infoleak to defeat KASLR this allows for arbitrary code to be executed in kernel mode. All Vita devices and firmwares before 3.61 are affected.

The bug's present in `sceNetIoctl` function. Let's go.

Implementation

Let's take a look at how `sceNetIoctl(s, flags, umem)` works. I'll be providing links to `ioctl` from NetBSD because it looks very similar; and there also will be some pseudo-code which does not match Vita version exactly, but is close enough.

The arguments are:

- `s`: socket id/descriptor
- `flags`: ioctl number
- `umem`: pointer to user memory

First it checks that the kernel heap has enough memory (at least 5 free blocks of size 0x800 bytes or more) and then calls another function which is the real meat of the syscall.

```
if ( check_heap_space_available(0x800, 5) )
        res = ioctl(s, flags, umem);
else
        res = -55;
```

A few words about how SceNetPs works. One of the first things most network-related syscalls do is lock a global SceNetPs mutex (which I am going to call `g_network_mutex`). Indeed, instead of fine-grained locking it just has a global mutex, which is probably terrible for performance, but I digress.

So the next step of our `ioctl` implementation is locking `g_network_mutex`. After that, a pointer to the socket is retrieved by its descriptor from the global socket table.

```
sce_psnet_bnet_mutex_lock(&g_network_mutex, 0);
socket = get_socket_by_id(s, &ret, 0);
```

Next, a buffer for copying from/to user is allocated. The logic is same as in NetBSD. Notice the `KM_SLEEP` flag here: it means that alloc will never return `NULL`, it guarantees success by waiting/rescheduling the thread until more kernel memory becomes available.

```
memsz = (flags >> 16) & 0x1FFF;
if ( memsz > 0x80 ) {
        heapmem = malloc(memsz, 0, 8); // 0 here is equivalent to KM_SLEEP
```

```
        mem = heapmem;
} else {
        heapmem = 0;
        mem = &stackbuf;
}
```

Following that, depending on `flags`, either user data is copied in or the buffer is zeroed.

```
if ( (flags & 0x80000000) && memsz )
        ret = u2k_copy(umem, mem, memsz); // args = src, dst, size
else if ( (flags & 0x40000000) && memsz )
        memset(mem, 0, memsz);
else if ( flags & 0x20000000 )
        *(void **)mem = umem;
```

Then, a function with ID `id = (flags & 0xFF00) >> 8` is executed. There are two hardcoded IDs and if there's no match, a generic function from socket vtable is executed, in pseudocode with missing casts:

```
vptr = *(uint32_t*)(socket + 24);
fptr = *(uint32_t*)(vptr + 28);
err = fptr(socket, 11, flags, kmem);
        |
        +-------------------------------------------+
                                                    |
// Spoiler Alert: we're going to gain control over this pointer
```

We're at the end now, time to free allocated heap memory, unlock the mutex and return the result error code to the caller:

```
if ( !err && (flags & 0x40000000) && memsz)
    err = k2u_copy(mem, umem, memsz); // args = src, dst, size

free(heapmem);
sce_psnet_bnet_mutex_unlock(&g_network_mutex);
return err;
```

Now that we're done with the boring description of how SceNetPs ioctls work, let's get to the fun part.

The bug

There are multiple problems in this code that eventually lead to code execution:

1. **no reference counting!** what if our poor socket gets freed during the operation of ioctl? This is the real bug which results in UaF, however, it helps that:
2. the heap free check is insufficient: it checks for five blocks sized >=0x800 but we can cause an allocation of 0x1FFF bytes at most
3. when malloc function is unable to fulfill a request, instead of returning NULL, it will wait for more memory to become available (due to Vita analogue of the `KM_SLEEP` param passed to it)

Of course, before malloc goes to sleep, it also unlocks the `g_network_mutex`.

N.B. the bug is strictly Vita-only, other implementations (NetBSD, OpenBSD) do proper reference counting.

Exploit

So here's our exploitation strategy:

1. thread #1: Allocate a socket
2. thread #1: Fill up `SceNetPs` heap memory
3. thread #1: Call an `ioctl` on the socket. Now this thread will wait for heap memory to become available
4. thread #2: Free the socket
5. thread #2: Overwrite freed socket memory with controlled data
6. thread #2: Free `SceNetPs` heap memory so that the first thread wakes up.
7. thread #1: Code exec!

allocate a socket

This is harder than it sounds. The reason is that SceNetPs uses a memory pool, a single-linked list of free blocks of the same size, to allocate sockets. However:

- When the pool is full, instead of adding a freed socket's memory to the pool, it calls `free()` on it.
- When the pool is empty, instead of allocating new socket from the pool, it calls `malloc()`.

Since we want to overwrite socket structure with controlled memory (i.e. with not-a-socket), we need to maintain these condition:

- before allocating our socket, empty the pool by allocating a lot of sockets
- before freeing the socket, fill up the pool by freeing a lot of sockets

fill up heap memory

The best primitive I was able to find that can be used to fill up the heap is a `dump`. For our exploit its real purpose doesn't matter. The `int sceNetDumpCreate(char *name, int len, int flags)` function eventually leads to a `dump = malloc(len + 0x40)` which is perfect for filling up the heap.

It should be noted that we also need to make at least 5 holes sized 0x800, in order for the heap free check to pass. This is achieved in the HENkaku exploit by allocating 10 dumps sized 0xF00 and then freeing even ones (0th, 2nd, 4th, 6th, …).

call an ioctl

Self-descriptive. Just call `sceNetIoctl` with proper arguments, making sure the ioctl number is invalid so that instead of a generic function, the `fptr` gets called, and `memsz` is 0x1FFF, the largest allowed value.

Since we've ensured there's not enough heap memory to allocate a single block of 0x1FFF bytes, this thread is now sleeping and waiting for more kernel heap memory to become available.

free the socket

Now that the first thread has fallen asleep, unlocking `g_network_mutex` in the process, the second thread can free the socket the ioctl is operating on right now.

overwrite memory with controlled data

For that, I've used `sceNetControl(int a1, int a2, void *ubuf, int argsize)`. When passed proper flags, it will allocate a `buffer = malloc(argsize)` and then copy in user data. At the end of the function, the `buffer` is freed but it doesn't really matter because `free()` doesn't destroy our data.

With an `argsize` that matches the socket object size and proper kernel heap spraying, the allocated buffer will have the same address as the freed socket.

free heap memory

Just free some of the dumps allocated in the second step. When there's a contiguous block of 0x1FFF free bytes available, the first thread will wake up and give us …

code exec

Since we control the whole socket struct, we can set `vptr` to whatever value we'd like, and as such gain control over `fptr` as well. However! There are some exploit mitigations that we still have to bypass.

Bypassing the mitigations

KASLR & SMEP/SMAP-like

Vita's architecture is ARM which doesn't have anything named SMEP or SMAP, however these terms are widely used and understood. (On Vita such functionality is implemented using the Domain Access Control Register).

Basically:

- the kernel cannot execute user executable code
- the kernel cannot read or write user memory (outside of utility functions specifically made for that purpose)

As a result, we cannot just point `vptr` into user memory, or `fptr` into user code.

We bypass both KASLR & DACR protections with one kernel stack disclosure in the `sceIoDevctl` system call:

- to bypass KASLR, leak a `SceSysmem` address
- to bypass DACR, leak kernel thread stack base. Then plant our data into the kernel stack

The same `sceIoDevctl` system call with different arguments is used for planting data into the kernel stack.

XN/NX

Now that we have our data in the kernel space, to bypass the execute never bit (only executable memory can be executed), build a ROP chain in kernel space and pivot to it.

Since we've only leaked SceSysmem base, we can only use gadgets from that kernel module, but that's more than enough.

That's it

You can find source code for the exploit here. Bring your own kernel ROP chain.