Hacking the PS4, part 1

Introduction to PS4's security, and userland ROP

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

Introduction

Since there haven't been any major public announcements regarding PS4 hacking for a long time now, I wanted to explain a bit about how far PS4 hacking has come, and what is preventing further progression.

I will explain some security concepts that generally apply to all modern systems, and the discoveries that I have made from running ROP tests on my PS4.

The goal of this series will be to present a full chain of exploits to ultimately gain kernel code execution on the PS4 by just visiting a web page on the Internet Browser.

If you are not particularly familiar with exploitation, you should read my article about exploiting DS games through stack smash vulnerabilities in save files first.

You may download my complete setup here to run these tests yourself; it is currently for firmware 1.76 only. If you are on an older firmware and wish to update to 1.76, you may download the 1.76 PUP file and update via USB.

Background information about the PS4

> As you probably know the PS4 features a custom AMD x86-64 CPU (8 cores), and there are loads of research available for this CPU architecture, even if this specific version might deviate slightly from known standards. For example, PFLA (Page Fault Liberation Army) released a proof of concept implementing a complete Turing machine using only page faults and the x86 MMU during the 29C3 congress, check their awesome video over at YouTube. Also interesting if you are trying to run code within a virtual machine and want to execute instructions on the host CPU.

- EurAsia news article 3251

As well as having a well documented CPU architecture, much of the software used in the PS4 is open source.

Most notably, the PS4's Orbis OS is based on FreeBSD (9.0), just like the PS3's OS was (with parts of NetBSD as well); and includes a wide variety of additional open source software as well, such as Mono VM, and WebKit.

WebKit entry point

WebKit is the open source layout engine which renders web pages in the browsers for iOS, Wii

U, 3DS, PS Vita, and the PS4.

Although so widely used and mature, WebKit does have its share of vulnerabilities; you can learn about many of them by reading Pwn2Own write-ups.

In particular, the browser in PS4 firmware 1.76 uses a version of WebKit which is vulnerable to CVE-2012-3748, a heap-based buffer overflow in the `JSArray::sort(...)` method.

In 2014 nas and Proxima announced that they had successfully been able to port an exploit using this vulnerability, originally written for Mac OS X Safari, to the PS4's internet browser, and released the PoC code publicly as the first entry point into hacking the PS4.

This gives us arbitrary read and write access to everything the WebKit process can read and write to, which can be used to dump modules, and overwrite return addresses on the stack, letting us control the instruction pointer register (`rip`) to achieve ROP execution.

Since then, many other vulnerabilities have been found in WebKit, which could probably be used as an entry point for later firmwares of the PS4, but as of writing, no one has ported any of these exploits to the PS4 publicly.

If you have never signed into PSN, your PS4 won't be able to open the Internet Browser, however you can go to "Settings", and then "User's Guide" to open a limited web browser view which you can control the contents of with a proxy.

What is ROP?

Unlike in primitive devices like the DS, the PS4 has a kernel which controls the properties of different areas of memory. Pages of memory which are marked as executable cannot be overwritten, and pages of memory which are marked as writable cannot be executed; this is known as Data Execution Prevention (DEP).

This means that we can't just copy a payload into memory and execute it. However, we can execute code that is already loaded into memory and marked as executable.

It wouldn't be very useful to jump to a single address if we can't write our own code to that address, so we use ROP.

Return-Oriented Programming (ROP) is just an extension to traditional stack smashing, but instead of overwriting only a single value which `rip` will jump to, we can chain together many different addresses, known as gadgets.

A gadget is usually just a single desired instruction followed by a `ret`.

In x86_64 assembly, when a `ret` instruction is reached, a 64 bit value is popped off the stack and `rip` jumps to it; since we can control the stack, we can make every `ret` instruction jump to the next desired gadget.

For example, from `0x80000` may contains instructions:

```
mov rax, 0
ret
```

And from `0x90000` may contain instructions:

```
mov rbx, 0
ret
```

If we overwrite a return address on the stack to contain `0x80000` followed by `0x90000`, then as soon as the first `ret` instruction is reached execution will jump to `mov rax, 0`, and immediately afterwards, the next `ret` instruction will pop `0x90000` off the stack and jump to `mov rbx, 0`.

Effectively this chain will set both `rax` and `rbx` to 0, just as if we had written the code into a single location and executed it from there.

ROP chains aren't just limited to a list of addresses though; assuming that from `0xa0000` contains these instructions:

```
pop rax
ret
```

We can set the first item in the chain to `0xa0000` and the next item to any desired value for `rax`.

Gadgets also don't have to end in a `ret` instruction; we can use gadgets ending in a `jmp`:

```
add rax, 8
jmp rcx
```

By making `rcx` point to a `ret` instruction, the chain will continue as normal:

```
chain.add("pop rcx", "ret");
chain.add("add rax, 8; jmp rcx");
```

Sometimes you won't be able to find the exact gadget that you need on its own, but with other instructions after it. For example, if you want to set `r8` to something, but only have this gadget, you will have to set `r9` to some dummy value:

```
pop r8
pop r9
ret
```

Although you may have to be creative with how you write ROP chains, it is generally accepted that within a sufficiently large enough code dump, there will be enough gadgets for Turing-complete functionality; this makes ROP a viable method of defeating DEP.

Finding gadgets

Think of ROP as writing a new chapter to a book, using only words that have appeared at the end of sentences in the previous chapters.

It's obvious from the structure of most sentences that we probably won't be able to find words like 'and' or 'but' appearing at the end of any sentences, but we will need these connectives in order to write anything meaningful.

It is quite possible however, that a sentence has ended with 'sand'. Although the author only ever intended for the word to be read from the 's', if we start reading from the 'a', it will appear as an entirely different word by coincidence, 'and'.

These principles also apply to ROP.

Since almost all functions are structured with a prologue and epilogue:

```
; Save registers
push    rbp
mov     rbp, rsp
push    r15
push    r14
push    r13
push    r12
push    rbx
sub     rsp, 18h

; Function body

; Restore registers
add     rsp, 18h
pop     rbx
pop     r12
pop     r13
pop     r14
pop     r15
pop     rbp
ret
```

You'd expect to only be able to find `pop` gadgets, or more rarely, something like `xor rax, rax` to set the return value to 0 before returning.

Having a comparison like:

```
cmp [rax], r12
ret
```

Wouldn't make any sense since the result of the comparison isn't used by the function. However, there is still a possibility that we can find gadgets like these.

x86_64 instructions are similar to words in that they have variable lengths, and can mean something entirely different depending on where decoding starts.

> The x86_64 architecture is a variable-length CISC instruction set. Return-oriented programming on the x86_64 takes advantage of the fact that the instruction set is very "dense", that is, any random sequence of bytes is likely to be interpretable as some valid set of x86_64 instructions.

- *Wikipedia*

To demonstrate this, take a look at the end of this function from the WebKit module:

```
000000000052BE0D                mov     eax, [rdx+8]
000000000052BE10                mov     [rsi+10h], eax
000000000052BE13                or      byte ptr [rsi+39h], 20h
000000000052BE17                ret
```

Now take a look at what the code looks like if we start decoding from `0x52be14`:

```
000000000052BE14                cmp     [rax], r12
000000000052BE17                ret
```

Even though this code was never intended to be executed, it is within an area of memory which has been marked as executable, so it is perfectly valid to use as a gadget.

Of course, it would be incredibily time consuming to look at every possible way of interpreting code before every single `ret` instruction manually; and that's why tools exist to do this for you. The one which I use to search for ROP gadgets is rp++; to generate a text file filled with gadgets, just use:

```
rp-win-x64 -f mod14.bin --raw=x64 --rop=1 --unique > mod14.txt
```

General protection faults

If we *do* perform an access violation, such as by trying to execute a non-executable page of memory, or by trying to write to a non-writable page of memory, a general protection fault, or more specifically in this instance, a segmentation fault, will occur.

For example, trying to execute code on the stack, which is mapped as read and write only:

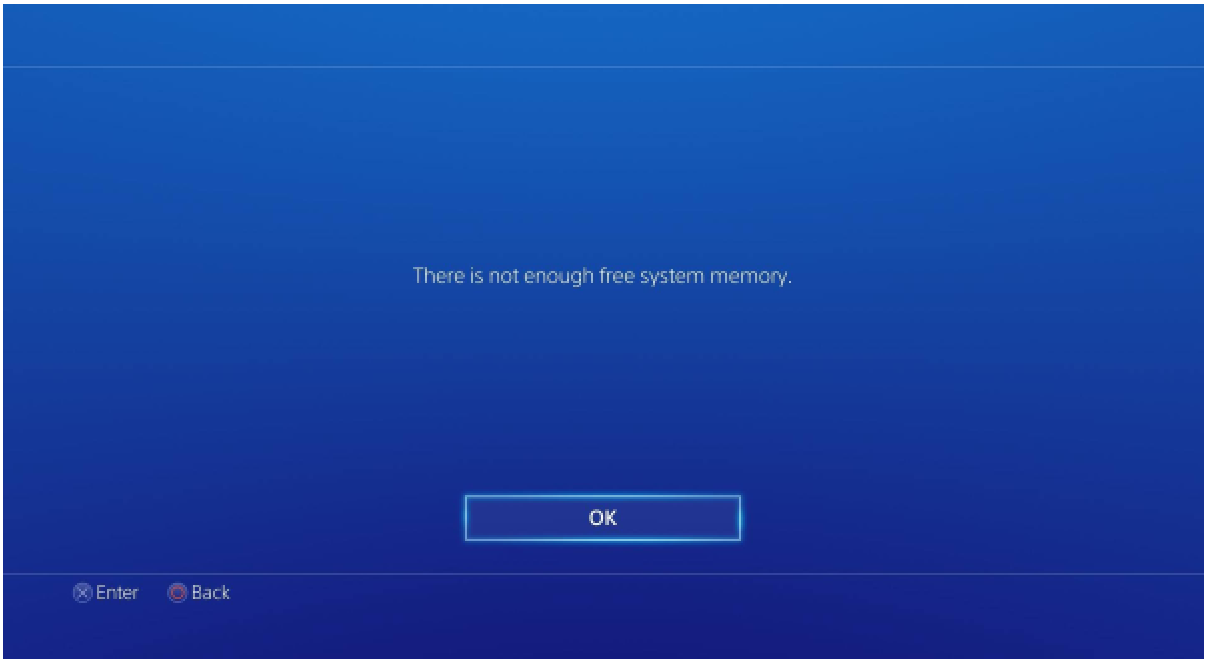
```
setU8to(chain.data + 0, 0xeb);
setU8to(chain.data + 1, 0xfe);

chain.add(chain.data);
```

And trying to write to code, which is mapped as read and execute only:

```
setU8to(moduleBases[webkit], 0);
```

If a general protection fault occurs, a message saying "There is not enough free system memory" will appear, and the page will fail to load:



This message will also be displayed for other hard faults, such as division by 0, or execution of an invalid instruction or unimplemented system call, but most commonly it will be encountered by performing a segmentation fault.

ASLR

Address Space Layout Randomization (ASLR) is a security technique which causes the base addresses of modules to be different every time you start the PS4.

It has been reported to me that very old firmwares (1.05) don't have ASLR enabled, but it was introduced sometime before firmware 1.70. Note that kernel ASLR is not enabled (for firmwares 1.76 and lower at least), which will be proved later in the article.

For most exploits ASLR would be a problem because if you don't know the addresses of the gadgets in memory, you would have no idea what to write to the stack.

Luckily for us, we aren't limited to just writing static ROP chains. We can use JavaScript to read the modules table, which will tell us the base addresses of all loaded modules. Using these bases, we can then calculate the addresses of all our gadgets before we trigger ROP execution, defeating ASLR.

The modules table also includes the filenames of the modules:

- WebProcess.self
- libkernel.sprx
- libSceLibcInternal.sprx
- libSceSysmodule.sprx
- libSceNet.sprx
- libSceNetCtl.sprx
- libSceIpmi.sprx
- libSceMbus.sprx
- libSceRegMgr.sprx
- libSceRtc.sprx
- libScePad.sprx
- libSceVideoOut.sprx
- libScePigletv2VSH.sprx
- libSceOrbisCompat.sprx
- libSceWebKit2.sprx

- libSceSysCore.sprx
- libSceSsl.sprx
- libSceVideoCoreServerInterface.sprx
- libSceSystemService.sprx
- libSceCompositeExt.sprx

Although the PS4 predominantly uses the [**S**igned] **P**PU **R**elocatable E**x**ecutable ([S]PRX) format for modules, some string references to [**S**igned] **E**xecutable and **L**inking **F**ormat ([S]ELF) object files can also be found in the `libSceSysmodule.sprx` dump, such as `bdj.elf`, `web_core.elf` and `orbis-jsc-compiler.self`. This combination of modules and objects is similar to what is used in the PSP and PS3.

You can view a complete list of all modules available (not just those loaded by the browser) in `libSceSysmodule.sprx`. We can load and dump some of these through several of Sony's custom system calls, which will be explained later in this article.

JuSt-ROP

Using JavaScript to write and execute dynamic ROP chains gives us a tremendous advantage over a traditional, static buffer overflow attack.

As well as being necessary to defeat ASLR, JavaScript also lets us read the user agent of the browser, and provide different ROP chains for different browser versions, giving our exploit a greater range of compatibility.

We can even use JavaScript to read the memory at our gadgets' addresses to check that they are correct, giving us almost perfect reliability. Theoretically, you could take this even further by writing a script to dynamically find ROP gadgets and then build ROP chains on the fly.

Writing ROP chains dynamically, rather than generating them with a script beforehand, just makes sense.

I created a JavaScript framework for writing ROP chains, JuSt-ROP, for this very reason.

JavaScript caveats

JavaScript represents numbers using the IEEE-754 double-precision (64 bit) format. This provides us with 53 bit precision, meaning that it isn't possible to represent every 64 bit value, approximations will have to be used for some.

If you just need to set a 64 bit value to something low, like `256`, then `setU64to` will be fine.

But for situations in which you need to write a buffer or struct of data, there is the possibility that certain bytes will be written incorrectly if it has been written in 64 bit chunks.

Instead, you should write data in 32 bit chunks (remembering that the PS4 is little endian), to ensure that every byte is exact.

System calls

Interestingly, the PS4 uses the same calling convention as Linux and MS-DOS for system calls, with arguments stored in registers, rather than the traditional UNIX way (which FreeBSD uses by default), with arguments stored in the stack:

- `rax` - System call number
- `rdi` - Argument 1

- `rsi` - Argument 2
- `rdx` - Argument 3
- `r10` - Argument 4
- `r8` - Argument 5
- `r9` - Argument 6

We can try to perform any system call with the following JuSt-ROP method:

```
this.syscall = function(name, systemCallNumber, arg1, arg2, arg3, arg
    console.log("syscall " + name);

    this.add("pop rax", systemCallNumber);
    if(typeof(arg1) !== "undefined") this.add("pop rdi", arg1);
    if(typeof(arg2) !== "undefined") this.add("pop rsi", arg2);
    if(typeof(arg3) !== "undefined") this.add("pop rdx", arg3);
    if(typeof(arg4) !== "undefined") this.add("pop rcx", arg4);
    if(typeof(arg5) !== "undefined") this.add("pop r8", arg5);
    if(typeof(arg6) !== "undefined") this.add("pop r9", arg6);
    this.add("mov r10, rcx; syscall");
}
```

Just make sure to set the stack base to some free memory beforehand:

```
this.add("pop rbp", stackBase + returnAddress + 0x1400);
```

Using system calls can tell us a huge amount about the PS4 kernel. Not only that, but using system calls is most likely the only way that we can interact with the kernel, and thus potentially trigger a kernel exploit.

If you are reverse engineering modules to identify some of Sony's custom system calls, you may come across an alternative calling convention:

Sometimes Sony performs system calls through regular system call 0 (which usually does nothing in FreeBSD), with the first argument (`rdi`) controlling which system call should be executed:

- `rax` - 0
- `rdi` - System call number
- `rsi` - Argument 1
- `rdx` - Argument 2
- `r10` - Argument 3
- `r8` - Argument 4
- `r9` - Argument 5

It is likely that Sony did this to have easy compatibility with the function calling convention. For example:

```
.global syscall
syscall:
```

```
    xor     rax, rax
    mov     r10, rcx
    syscall
    ret
```

Using this, they can perform system calls from C using the function calling convention:

```
int syscall();

int getpid(void) {
    return syscall(20);
}
```

When writing ROP chains, we can use either convention:

```
// Both will get the current process ID:
chain.syscall("getpid", 20);
chain.syscall("getpid", 0, 20);
```

It's good to be aware of this, because we can use whichever one is more convenient for the gadgets that are available.

getpid

Just by using system call 20, getpid(void), we can learn a lot about the kernel.

The very fact that this system call works at all tells us that Sony didn't bother mixing up the system call numbers as a means of security through obscurity (under the BSD license they could have done this without releasing the new system call numbers).

So, we automatically have a list of system calls in the PS4 kernel to try.

Secondly, by calling `getpid()`, restarting the browser, and calling it again, we get a return value 2 higher than the previous value.

This tells us that the Internet Browser app actually consists of 2 separate processes: the WebKit core (which we take over), that handles parsing HTML and CSS, decoding images, and executing JavaScript for example, and another one to handle everything else: displaying graphics, receiving controller input, managing history and bookmarks, etc.

Also, although FreeBSD has supported PID randomisation since 4.0, sequential PID allocation is the default behaviour.

The fact that PID allocation is set to the default behaviour indicates that Sony likely didn't bother adding any additional security enhancements such as those encouraged by projects like HardenedBSD, other than userland ASLR.

How many custom system calls are there?

The last standard FreeBSD 9 system call is `wait6`, number `532`; anything higher than this must be a custom Sony system call.

Invoking most of Sony's custom system calls without the correct arguments will return error `0x16`, `"Invalid argument"`; however, any compatibility or unimplemented system calls will report the "There is not enough free system memory" error.

Through trial and error, I have found that system call number `617` is the last Sony system call, anything higher is unimplemented.

From this, we can conclude that there are 85 custom Sony system calls in the PS4's kernel (617 - 532).

libkernel.sprx

To identify how custom system calls are used by libkernel, you must first remember that it is just a modification of the standard FreeBSD 9.0 libraries.

Here's an extract of `_libpthread_init` from thr_init.c:

```
/*
 * Check for the special case of this process running as
 * or in place of init as pid = 1:
 */
if ((_thr_pid = getpid()) == 1) {
    /*
     * Setup a new session for this process which is
     * assumed to be running as root.
     */
    if (setsid() == -1)
        PANIC("Can't set session ID");
    if (revoke(_PATH_CONSOLE) != 0)
        PANIC("Can't revoke console");
    if ((fd = __sys_open(_PATH_CONSOLE, O_RDWR)) < 0)
        PANIC("Can't open console");
    if (setlogin("root") == -1)
        PANIC("Can't set login to root");
    if (_ioctl(fd, TIOCSCTTY, (char *) NULL) == -1)
        PANIC("Can't set controlling terminal");
}
```

The same function can be found at offset `0x215F0` from `libkernel.sprx`. This is how the above extract looks from within a libkernel dump:

```
call    getpid
mov     cs:dword_5B638, eax
cmp     eax, 1
jnz     short loc_2169F
```

```
call    setsid
cmp     eax, 0FFFFFFFFh
jz      loc_21A0C

lea     rdi, aDevConsole ; "/dev/console"
call    revoke
test    eax, eax
jnz     loc_21A24

lea     rdi, aDevConsole ; "/dev/console"
mov     esi, 2
xor     al, al
call    open

mov     r14d, eax
test    r14d, r14d
js      loc_21A3C
lea     rdi, aRoot       ; "root"
call    setlogin
cmp     eax, 0FFFFFFFFh
jz      loc_21A54

mov     edi, r14d
mov     esi, 20007461h
xor     edx, edx
xor     al, al
call    ioctl
cmp     eax, 0FFFFFFFFh
jz      loc_21A6C
```

Reversing module dumps to analyse system calls

libkernel isn't completely open source though; there's also a lot of custom code which can help disclose some of Sony's system calls.

Although this process will vary depending on the system call you are looking up; for some, it is fairly easy to get a basic understanding of the arguments that are passed to it.

The system call wrapper will be declared somewhere in `libkernel.sprx`, and will almost always follow this template:

```
000000000000DB70 syscall_601     proc near
000000000000DB70                 mov     rax, 259h
000000000000DB77                 mov     r10, rcx
000000000000DB7A                 syscall
000000000000DB7C                 jb      short error
```

```
000000000000DB7E                        retn
000000000000DB7F
000000000000DB7F error:
000000000000DB7F                        lea     rcx, sub_DF60
000000000000DB86                        jmp     rcx
000000000000DB86 syscall_601            endp
```

Note that the `mov r10, rcx` instruction doesn't necessarily mean that the system call takes at least 4 arguments; all system call wrappers have it, even those that take no arguments, such as `getpid`.

Once you've found the wrapper, you can look up xrefs to it:

```
0000000000011D50                        mov     edi, 10h
0000000000011D55                        xor     esi, esi
0000000000011D57                        mov     edx, 1
0000000000011D5C                        call    syscall_601
0000000000011D61                        test    eax, eax
0000000000011D63                        jz      short loc_11D6A
```

It's good to look up several of these, just to make sure that the registers weren't modified for something unrelated:

```
0000000000011A28                        mov     edi, 9
0000000000011A2D                        xor     esi, esi
0000000000011A2F                        xor     edx, edx
0000000000011A31                        call    syscall_601
0000000000011A36                        test    eax, eax
0000000000011A38                        jz      short loc_11A3F
```

Consistently, the first three registers of the system call convention (`rdi`, `rsi`, and `rdx`) are modified before invoking the call, so we can conclude with reasonable confidence that it takes 3 arguments.

For clarity, this is how we would replicate the calls in JuSt-ROP:

```
chain.syscall("unknown", 601, 0x10, 0, 1);
chain.syscall("unknown", 601, 9, 0, 0);
```

As with most system calls, it will return 0 on success, as seen by the `jz` conditional after `test`ing the return value.

Looking up anything beyond than the amount of arguments will require a much more in-depth analysis of the code before and after the call to understand the context, but this should help you get started.

Brute forcing system calls

Although reverse engineering module dumps is the most reliable way to identify system calls, some aren't referenced at all in the dumps we have so we will need to analyse them blindly.

If we guess that a certain system call might take a particular set of arguments, we can brute force all system calls which return a certain value (`0` for success) with the arguments that we chose, and ignore all which returned an error.

We can also pass 0s for all arguments, and brute force all system calls which return useful errors such as `0xe`, `"Bad address"`, which would indicate that they take at least one pointer.

Firstly, we will need to execute the ROP chain as soon as the page loads. We can do this by attaching our function to the `body` element's `onload`:

```
<body onload="exploit()">
```

Next we will need to perform a specific system call depending on an HTTP GET value. Although this can be done with JavaScript, I will demonstrate how to do this using PHP for simplicity:

```
var Sony = 533;
chain.syscall("Sony system call", Sony + <?php print($_GET["b"]); ?>,
chain.write_rax_ToVariable(0);
```

Once the system call has executed, we can check the return value, and if it isn't interesting, redirect the page to the next system call:

```
if(chain.getVariable(0) == 0x16) window.location.assign("index.php?b=
```

Running the page with `?b=0` appended to the end will start the brute force from the first Sony system call.

Although this method requires a lot of experimentation, by passing different values to some of the system calls found by brute forcing and analysing the new return values, there are a few system calls which you should be able to partially identify.

System call 538

As an example, I'll take a look at system call 538, without relying on any module dumps.

These are the return values depending on what is passed as the first argument:

- 0 - `0x16`, `"Invalid argument"`
- 1 - `0xe`, `"Bad address"`
- Pointer to 0s - `0x64` initially, but each time the page is refreshed this value increases by 1

Other potential arguments to try would be PID, thread ID, and file descriptor.

Although most system calls will return `0` on success, due to the nature of the return value increasing after each time it is called, it seems like it is allocating a resource number, such as a file descriptor.

The next thing to do would be to look at the data before and after performing the system call, to see if it has been written to.

Since there is no change in the data, we can assume that it is an input for now.

I then tried passing a long string as the first argument. You should always try this with every input you find because there is the possibility of discovering a buffer overflow.

```
writeString(chain.data, "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa");
chain.syscall("unknown", 538, chain.data, 0, 0, 0, 0, 0);
```

The return value for this is `0x3f`, `ENAMETOOLONG`. Unfortunately it seems that this system call correctly limits the name (32 bytes including `NULL` truncator), but it does tell us that it *is* expecting a string, rather than a struct.

We now have a few possibilities for what this system call is doing, the most obvious being something related to the filesystem (such as a custom `mkdir` or `open`), but this doesn't seem particularly likely seeing as a resource was allocated even before we wrote any data to the pointer.

To test whether the first parameter is a path, we can break it up with multiple `/` characters to see if this allows for a longer string:

```
writeString(chain.data, "aaaaaaaaaa/aaaaaaaaaa/aaaaaaaaaa");
chain.syscall("unknown", 538, chain.data, 0, 0, 0, 0, 0);
```

Since this also returns `0x3f`, we can assume that the first argument isn't a path; it is a name for *something* that gets allocated a sequential identifier.

After analysing some more system calls, I found that the following all shared this exact same behaviour:

- 533
- 538
- 557
- 574
- 580

From the information that we have so far, it is almost impossible to pinpoint exactly what these system calls do, but as you run more tests, further information will slowly be revealed.

To save you some time, system call 538 is allocating an event flag (and it doesn't just take a name).

Using general knowledge of how a kernel works, you can guess, and then verify, what the system calls are allocating (semaphores, mutexes, etc).

Dumping additional modules

We can dump additional modules by following these stages:

- Load the module
- Get the module's base address
- Dump the module

I've extracted and posted a list of all module names on psdevwiki.

To load a module we will need to use the `sceSysmoduleLoadModule` function from `libSceSysmodule.sprx + 0x1850`. The first parameter is the module ID to load, and the other 3 should just be passed 0.

The following JuSt-ROP method can be used to perform a function call:

```
this.call = function(name, module, address, arg1, arg2, arg3, arg4, a
    console.log("call " + name);

    if(typeof(arg1) !== "undefined") this.add("pop rdi", arg1);
    if(typeof(arg2) !== "undefined") this.add("pop rsi", arg2);
    if(typeof(arg3) !== "undefined") this.add("pop rdx", arg3);
    if(typeof(arg4) !== "undefined") this.add("pop rcx", arg4);
    if(typeof(arg5) !== "undefined") this.add("pop r8", arg5);
    if(typeof(arg6) !== "undefined") this.add("pop r9", arg6);
    this.add(module_bases[module] + address);
}
```

So, to load `libSceAvSetting.sprx` (`0xb`):

```
chain.call("sceSysmoduleLoadModule", libSysmodule, 0x1850, 0xb, 0, 0,
```

Unfortunately, a fault will be triggered when trying to load certain modules; this is because the `sceSysmoduleLoadModule` function doesn't load dependencies, so you will need to manually load them first.

Like most system calls, this should return `0` on success. To see the loaded module ID that was allocated, we can use one of Sony's custom system calls, number `592`, to get a list of currently loaded modules:

```
var countAddress = chain.data;
var modulesAddress = chain.data + 8;

// System call 592, getLoadedModules(int *destinationModuleHandles, i
chain.syscall("getLoadedModules", 592, modulesAddress, 256, countAddr

chain.execute(function() {
    var count = getU64from(countAddress);
    for(var index = 0; index < count; index++) {
        logAdd("Module handle: 0x" + getU32from(modulesAddress + inde
    }
});
```

```
});
```

Running this without loading any additional modules will produce the following list:

```
0x0, 0x1, 0x2, 0xc, 0xe, 0xf, 0x11, 0x12, 0x13, 0x14, 0x15, 0x16, 0x1
```

But if we run it after loading module `0xb`, we will see an additional entry, `0x65`. Remember that module ID is *not* the same as loaded module handle.

We can now use another of Sony's custom system calls, number `593`, which takes a module handle and a buffer, and fills the buffer with information about the loaded module, including its base address. Since the next available handle is always `0x65`, we can hardcode this value into our chain, rather than having to store the result from the module list.

The buffer must start with the size of the struct that should be returned, otherwise error `0x16` will be returned, `"Invalid argument"`:

```
setU64to(moduleInfoAddress, 0x160);
chain.syscall("getModuleInfo", 593, 0x65, moduleInfoAddress);

chain.execute(function() {
    logAdd(hexDump(moduleInfoAddress, 0x160));
});
```

It will return `0` upon success, and fill the buffer with a struct which can be read like so:

```
var name = readString(moduleInfoAddress + 0x8);
var codeBase = getU64from(moduleInfoAddress + 0x108);
var codeSize = getU32from(moduleInfoAddress + 0x110);
var dataBase = getU64from(moduleInfoAddress + 0x118);
var dataSize = getU32from(moduleInfoAddress + 0x120);
```

We now have everything we need to dump the module!

```
dump(codeBase, codeSize + dataSize);
```

There is another Sony system call, number `608`, which works in a similar way to `593`, but provides slightly different information about the loaded module:

```
setU64to(moduleInfoAddress, 0x1a8);
chain.syscall("getDifferentModuleInfo", 608, 0x65, 0, moduleInfoAddre
logAdd(hexDump(moduleInfoAddress, 0x1a8));
```

It's not clear what this information is.

It's not clear what this information is.

Browsing the filesystem

The PS4 uses the standard FreeBSD 9.0 system calls for reading files and directories.

However, whilst using `read` for some directories such as `/dev/` will work, others, such as `/` will fail.

I'm not sure why this is, but if we use `getdents` instead of `read` for directories, it will work much more reliably:

```
writeString(chain.data, "/dev/");
chain.syscall("open", 5, chain.data, 0, 0);
chain.write_rax_ToVariable(0);

chain.read_rdi_FromVariable(0);
chain.syscall("getdents", 272, undefined, chain.data + 0x10, 1028);
```

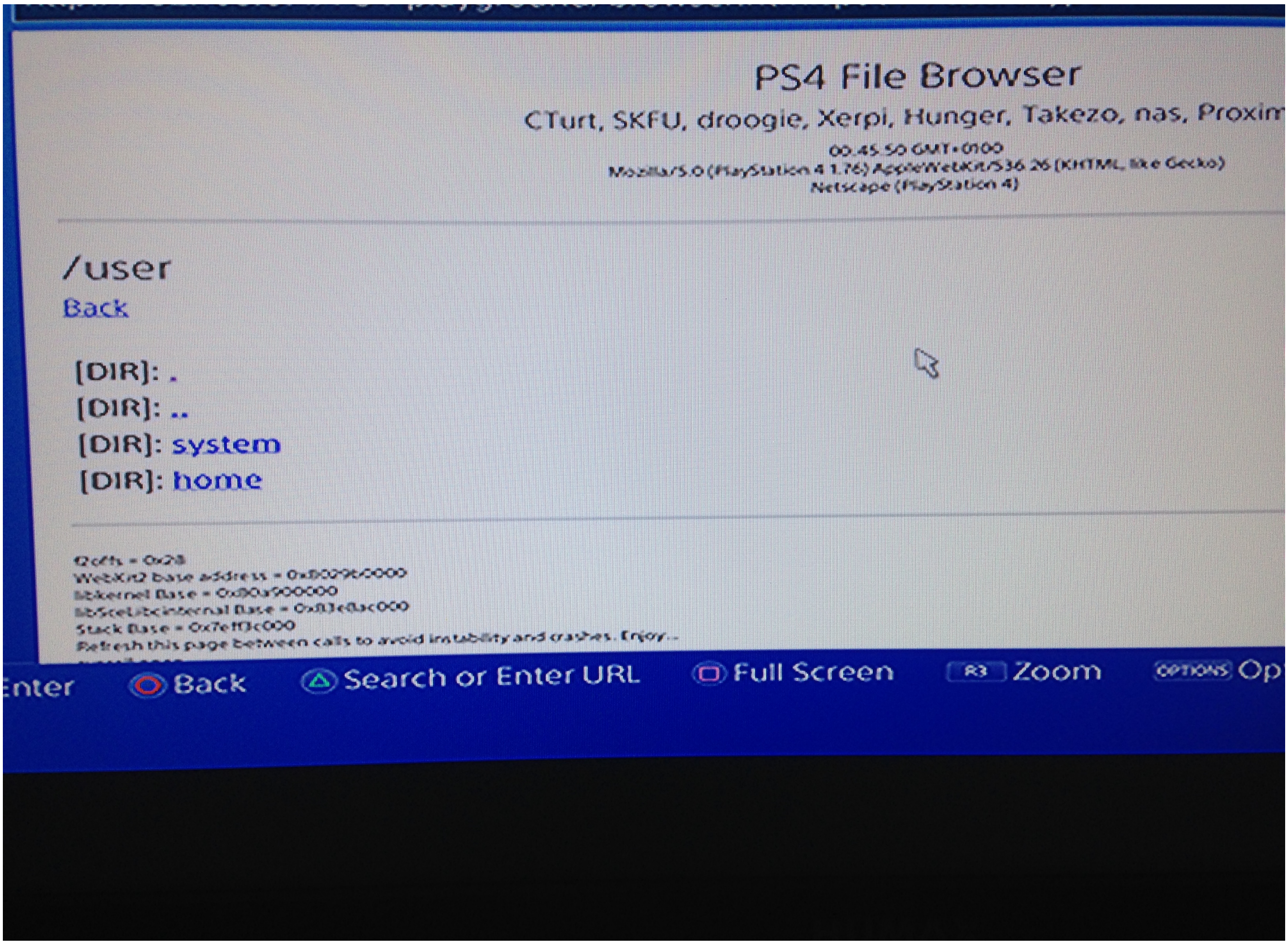
This is the resultant memory:

```
0000010: 0700 0000 1000 0205 6469 7073 7700 0000  ........dipsw...
0000020: 0800 0000 1000 0204 6e75 6c6c 0000 0000  ........null....
0000030: 0900 0000 1000 0204 7a65 726f 0000 0000  ........zero....
0000040: 0301 0000 0c00 0402 6664 0000 0b00 0000  ........fd......
0000050: 1000 0a05 7374 6469 6e00 0000 0d00 0000  ....stdin.......
0000060: 1000 0a06 7374 646f 7574 0000 0f00 0000  ....stdout......
0000070: 1000 0a06 7374 6465 7272 0000 1000 0000  ....stderr......
0000080: 1000 0205 646d 656d 3000 0000 1100 0000  ....dmem0.......
0000090: 1000 0205 646d 656d 3100 0000 1300 0000  ....dmem1.......
00000a0: 1000 0206 7261 6e64 6f6d 0000 1400 0000  ....random......
00000b0: 1000 0a07 7572 616e 646f 6d00 1600 0000  ....urandom.....
00000c0: 1400 020b 6465 6369 5f73 7464 6f75 7400  ....deci_stdout.
00000d0: 1700 0000 1400 020b 6465 6369 5f73 7464  ........deci_std
00000e0: 6572 7200 1800 0000 1400 0209 6465 6369  err.........deci
00000f0: 5f74 7479 3200 0000 1900 0000 1400 0209  _tty2...........
0000100: 6465 6369 5f74 7479 3300 0000 1a00 0000  deci_tty3.......
0000110: 1400 0209 6465 6369 5f74 7479 3400 0000  ....deci_tty4...
0000120: 1b00 0000 1400 0209 6465 6369 5f74 7479  ........deci_tty
0000130: 3500 0000 1c00 0000 1400 0209 6465 6369  5...........deci
0000140: 5f74 7479 3600 0000 1d00 0000 1400 0209  _tty6...........
0000150: 6465 6369 5f74 7479 3700 0000 1e00 0000  deci_tty7.......
0000160: 1400 020a 6465 6369 5f74 7479 6130 0000  ....deci_ttya0..
0000170: 1f00 0000 1400 020a 6465 6369 5f74 7479  ........deci_tty
0000180: 6230 0000 2000 0000 1400 020a 6465 6369  b0.. .......deci
0000190: 5f74 7479 6330 0000 2200 0000 1400 020a  _ttyc0..".......
00001a0: 6465 6369 5f73 7464 696e 0000 2300 0000  deci_stdin..#...
```

```
00001b0: 0c00 0203 6270 6600 2400 0000 1000 0a04  ....bpf.$.......
00001c0: 6270 6630 0000 0000 2900 0000 0c00 0203  bpf0....).......
00001d0: 6869 6400 2c00 0000 1400 0208 7363 655f  hid.,.......sce_
00001e0: 7a6c 6962 0000 0000 2e00 0000 1000 0204  zlib............
00001f0: 6374 7479 0000 0000 3400 0000 0c00 0202  ctty....4.......
0000200: 6763 0000 3900 0000 0c00 0203 6463 6500  gc..9.......dce.
0000210: 3a00 0000 1000 0205 6462 6767 6300 0000  :.......dbggc...
0000220: 3e00 0000 0c00 0203 616a 6d00 4100 0000  >.......ajm.A...
0000230: 0c00 0203 7576 6400 4200 0000 0c00 0203  ....uvd.B.......
0000240: 7663 6500 4500 0000 1800 020d 6e6f 7469  vce.E.......noti
0000250: 6669 6361 7469 6f6e 3000 0000 4600 0000  fication0...F...
0000260: 1800 020d 6e6f 7469 6669 6361 7469 6f6e  ....notification
0000270: 3100 0000 5000 0000 1000 0206 7573 6263  1...P.......usbc
0000280: 746c 0000 5600 0000 1000 0206 6361 6d65  tl..V.......came
0000290: 7261 0000 8500 0000 0c00 0203 726e 6700  ra..........rng.
00002a0: 0701 0000 0c00 0403 7573 6200 c900 0000  ........usb.....
00002b0: 1000 0a07 7567 656e 302e 3400 0000 0000  ....ugen0.4.....
00002c0: 0000 0000 0000 0000 0000 0000 0000 0000  ................
```

You can read some of these devices, for example: reading `/dev/urandom` will fill the memory with random data.

It is also possible to parse this memory to create a clean list of entries; look at `browser.html` in the repository for a complete file browser:



Unfortunately, due to sandboxing we don't have complete access to the file system. Trying to read files and directories that do exist but are restricted will give you error 2, `ENOENT`, `"No such file or directory"`.

We do have access to a lot of interesting stuff though including encrypted save data, trophies, and account information. I will go over more of the filesystem in my next article.

Sandboxing

As well as file related system calls failing for certain paths, there are other reasons for a system call to fail.

Most commonly, a disallowed system call will just return error `1`, `EPERM`, `"Operation not permitted"`; such as trying to use `ptrace`, but other system calls may fail for different reasons:

Compatibilty system calls are disabled. If you are trying to call `mmap` for example, you must use system call number 477, not 71 or 197; otherwise a segfault will be triggered.

Other system calls such as `exit` will also trigger a fault:

```
chain.syscall("exit", 1, 0);
```

Trying to create an SCTP socket will return error `0x2b`, `EPROTONOSUPPORT`, indicating that SCTP sockets have been disabled in the PS4 kernel:

```
//int socket(int domain, int type, int protocol);
//socket(AF_INET, SOCK_STREAM, IPPROTO_SCTP);
chain.syscall("socket", 97, 2, 1, 132);
```

And although calling `mmap` with `PROT_READ | PROT_WRITE | PROT_EXEC` will return a valid pointer, the `PROT_EXEC` flag is ignored. Reading its protection will return 3 (RW), and any attempt to execute the memory will trigger a segfault:

```
chain.syscall("mmap", 477, 0, 4096, 1 | 2 | 4, 4096, -1, 0);
chain.write_rax_ToVariable(0);
chain.read_rdi_FromVariable(0);
chain.add("pop rax", 0xfeeb);
chain.add("mov [rdi], rax");
chain.add("mov rax, rdi");
chain.add("jmp rax");
```

The list of open source software used in the PS4 doesn't list any kind of sandboxing software like Capsicum, so the PS4 must use either pure FreeBSD jails, or some kind of custom, proprietary, sandboxing system (unlikely).

Jails

We can prove the existence of FreeBSD jails being actively used in the PS4's kernel through the `auditon` system call being impossible to execute within a jailed environment:

```
chain.syscall("auditon", 446, 0, 0, 0);
```

The first thing the `auditon` system call does is check `jailed` here, and if so, return `ENOSYS`:

```
if (jailed(td->td_ucred))
    return (ENOSYS);
```

Otherwise the system call would most likely return `EPERM` from the `mac_system_check_auditon` here:

```
error = mac_system_check_auditon(td->td_ucred, uap->cmd);
if (error)
    return (error);
```

Or from the `priv_check` here:

```
error = priv_check(td, PRIV_AUDIT_CONTROL);
if (error)
    return (error);
```

The absolute furthest that the system call could reach would be immediately after the `priv_check`, here, before returning `EINVAL` due to the length argument being 0:

```
if ((uap->length <= 0) || (uap->length > sizeof(union auditon_udata))
    return (EINVAL);
```

Since `mac_system_check_auditon` and `priv_check` will never return `ENOSYS`, having the `jailed` check pass is the only way `ENOSYS` could be returned.

When executing the chain, `ENOSYS` *is* returned (`0x48`).

This tells us that whatever sandbox system the PS4 uses is at least based on jails because the `jailed` check passes.

FreeBSD 9.0 kernel exploits

Before trying to look for new vulnerabilities in the FreeBSD 9.0 kernel source code, we should first check whether any of the kernel vulnerabilities already found could be used on the PS4.

We can immediately dismiss some of these for obvious reasons:

- FreeBSD 9.0-9.1 mmap/ptrace - Privilege Escalation Exploit - this won't work since, as

- FreeBSD 9.0-9.1 mmap/ptrace - Privilege Escalation Exploit - this won't work since, as previously stated, we don't have access to the `ptrace` system call.
- FreeBSD 9.0 - Intel SYSRET Kernel Privilege Escalation Exploit - won't work because the PS4 uses an AMD processor.
- FreeBSD Kernel - Multiple Vulnerabilities - maybe the first vulnerability will lead to something, but the other 2 rely on SCTP sockets, which the PS4 kernel has disabled (as previously stated).

However, there are some smaller vulnerabilites, which could lead to something:

getlogin

One vulnerability which looks easy to try is using the getlogin system call to leak a small amount of kernel memory.

The `getlogin` system call is intended to copy the login name of the current session to userland memory, however, due to a bug, the whole buffer is always copied, and not just the size of the name string. This means that we can read some uninitialised data from the kernel, which might be of some use.

Note that the system call (49) is actually `int getlogin_r(char *name, int len);` and not `char *getlogin(void);`.

So, let's try copying some kernel memory into an unused part of userland memory:

```
chain.syscall("getlogin", 49, chain.data, 17);
```

Unfortunately 17 bytes is the most data we can get, since:

> Login names are limited to `MAXLOGNAME` (from `<sys/param.h>`) characters, currently 17 including null.

- *FreeBSD Man Pages*

After executing the chain, the return value was 0, which means that the system call worked! An excellent start. Now let's take a look at the memory which we pointed to:

Before executing the chain:

```
00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00
```

After executing the chain:

```
72 6f 6f 74 00 fe ff ff 08 62 61 82 ff ff ff ff
00
```

After decoding the first 4 bytes as ASCII:

```
root
```

So the browser is executed as root! That was unexpected.

But more interestingly, the memory leaked looks like a pointer to something in the kernel, which is always the same each time the chain is run; this is evidence to support Yifanlu's claims that the PS4 has no Kernel ASLR!

Summary

From the information currently available, the PS4's kernel seems to be very similar to the stock FreeBSD 9.0 kernel.

Importantly, the differences that *are* present appear to be from standard kernel configuration changes (such as disabling SCTP sockets), rather than from modified code. Sony have also added several of their own custom system calls to the kernel, but apart from this, the rest of the kernel seems fairly untouched.

In this respect, I'm inclined to believe that the PS4 shares most of the same juicy vulnerabilities as FreeBSD 9.0's kernel!

Unfortunately, most kernel exploits cannot be triggered from the WebKit entry point that we currently have due to sandboxing constraints (likely to be just stock FreeBSD jails).

And with FreeBSD 10 being out, it's unlikely that anyone is stashing away any private exploits for FreeBSD 9, so unless a new one is suddenly released, we're stuck with what is currently available.

The best approach from here seems to be reverse engineering all of the modules which can be dumped, in order to document as many of Sony's custom system calls as possible; I have a hunch that we will have more luck targeting these, than the standard FreeBSD system calls.

Recently Jaicrab has discovered two UART ports on the PS4 which shows us that there are hardware hackers interested in the PS4. Although the role of hardware hackers has traditionally been to dump the RAM of a system, like with the DSi, which we can already do thanks to the WebKit exploit, there's also the possibility of a hardware triggered kernel vulnerability being found, like geohot's original PS3 hypervisor hack. It remains most likely that a kernel exploit will be found on the PS4 through system call vulnerabilities though.

Thanks

- flatz
- SKFU
- droogie
- Xerpi
- bigboss
- Hunger
- Takezo
- Proxima