Hacking the PS4, part 2

Userland code execution

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 my first article on the PS4's security, I have made some new discoveries, aided by the fact that I now have code execution within the WebKit process.

Whilst I don't want to release my code execution solution yet, I have made my PS4-SDK open source, and will try to explain everything I have managed to do with it.

This article is less focused on exploitation, and more on what is possible with userland code execution under the WebKit process.

Update

After gaining kernel code execution, I've gone back to this article to update some of the uncertainties I had whilst researching with just userland code execution.

I've also since posted the method for gaining userland code execution, along with a ROP chain to load binaries sent over TCP; it is explained in part 3 of this series.

Code execution

As explained in my previous article, ROP is just executing existing code loaded in memory in a smart way; whilst ROP can technically be Turing-complete, it really isn't practical for anything more complex than some basic tests.

With the help of flatz, I've been able to leverage ROP to setup memory in such a way that I can write my own code into it, and execute it.

Simply, this means that I can compile C code, such as these examples included in PS4-SDK, and execute them as native x86_64 code.

Whilst this is big progress, we are still running within the Internet Browser, and have the same restrictions as before (like sandboxing).

As a little side note; with the recent release of LLVM 3.7, if we specify `-target x86_64-scei-ps4` to `clang`, we can compile code with the exact same options that Sony uses to compile official code for the PS4.

WebKit process limitations

As stated in my previous article, the Internet Browser actually consists of 2 separate

As stated in my previous article, the Internet Browser actually consists of 2 separate processes. The one which we hijack for code execution is the core WebKit process (which handles parsing HTML and CSS, decoding images, and executing JavaScript for example).

We can use the following code to dump all memory which our process has access to:

```
struct memoryRegionInfo info;
struct otherMemoryRegionInfo otherInfo;

void *m = NULL;

int i;

// Iterate over first 107 memory mappings
for(i = 0; i < 107; i++) {
    // Find base of next mapping
    getOtherMemoryInfo(m, 1, &otherInfo);

    // Get more info about this mapping
    getMemoryInfo(otherInfo.base, &info);

    // If readable, dump it
    if(info.flags & PROT_CPU_READ) {
        sceNetSend(sock, info.base, info.end - info.base, 0);
    }

    m = info.end;
}
```

Within this dump, you won't be able to find strings used by the other process, such as "Options", "Close Window", "Refresh", or "There is not enough free system memory".

Graphics

One of the main implications of this is clear: if the other process handles displaying graphics, we can't easily hijack the active `libSceVideoOut` handle.

I've been working with xerpi to try to reinitialise `libSceVideoOut`, but even though all functions are returning good values, we can't get the screen to change from the browser view.

Just to be certain that our process can't access any existing video handles created by the other process, we tried brute forcing all positive integers to see if any were valid.

Brute forcing with code execution

Brute forcing things with the ROP framework was very impractical. I relied on redirecting the page after each test, and since the exploit isn't 100% reliable, the brute forcer would get stuck after left for just a minute or so.

With real code execution, we can try to brute force more ambitious things, such as a video

With real code execution, we can try to brute force more ambitious things, such as a video handle that the Internet Browser has opened. And we can use sockets to track the progress remotely from a PC.

`sceVideoOutWaitVblank` will return an error if it is given an invalid handle, and 0 if it's given a valid handle:

```
int i;
for(i = 0; i < 0x7FFFFFFF; i++) {
    if(!sceVideoOutWaitVblank(i)) return i;
    if(i % 0x10000 == 0) debug(sock, "At %08x\n", i);
}

sceNetSocketClose(sock);

return 0;
```

After running this for several hours it returned 0, confirming that our process has no access to the other process' video handle.

Canvas

There is a partial solution to this though. If we create an HTML5 canvas and fill it with a single colour, we can find the address of its framebuffer in RAM, and create a new thread to render to it from native code, leaving the original thread to update the canvas as normal.

I've added an example of this to the PS4-SDK.

If the canvas has too high of a resolution, it is harder to locate its address, and will often have a poor refresh rate. However, we can stretch a low resolution image to be fullscreen, and it will work fine:

```
var body = document.getElementsByTagName("body")[0];

// Create canvas
var canvas = document.createElement("canvas");

canvas.id = "canvas";
canvas.width = 160;
canvas.height = 144;
canvas.style.zIndex = 1;
canvas.style.position = "absolute";
canvas.style.border = "1px solid";

// Centered
//canvas.style.left = ((window.screen.width - canvas.width) / 2).toSt
//canvas.style.top = ((window.screen.height - canvas.height) / 2).toS
```

```
// Fullscreen
canvas.style.left = "0px";
canvas.style.top = "0px";
canvas.style.width = "100%";
canvas.style.height = "100%";

body.appendChild(canvas);
```

Another thing you may want to do is remove all other elements before creating the canvas, as a slight performance boost, but also to prevent being able to scroll:

```
while(body.firstChild) {
    body.removeChild(body.firstChild);
}
```

And finally, you will want to hide the cursor:

```
document.body.style.cursor = "none";
```

Controller

The `libScePad` module is similar to `libSceVideoOut` in that it isn't used by our process, and so I wasn't able to get it working.

Calling `scePadOpen` will give an error, unless you call `scePadInit` beforehand. From this, we can tell that separate processes' modules each have their own internal state, and that our process wasn't using `libScePad` (since it wasn't already initialised).

So, like with graphics, we won't be able to hijack any handles already open, and trying to create new handles won't work either.

Maybe we can't read from the controller because it is already in use, and we would be able to read from a second controller, but unfortunately I can't test this since I only have 1 controller.

There are two workarounds for this: use the USB library to receive input from a third party controller, or just use any WiFi compatible device with buttons to send input over a UDP socket. I opted for using a Nintendo DS wirelessly.

USB flash drives

When you insert a USB into the PS4, a new device is listed under `/dev/`; `ugen0.4` for the first slot, and `ugen0.5` for the second slot.

Unfortunately, we can't mount the device since the `mount` system call (and variations like `nmount`) always return 1, `EPERM`.

However, we can access USB flash drives using the `libSceUsbd.sprx` module; it is very similar to `libusb`, but with the Sony naming convention, and the removal of contexts.

For example, the following `libusb` code:

```
libusb_context *context;
libusb_init(&context);
libusb_exit(context);
```

Would translate to this `libSceUsbd` code:

```
sceUsbdInit();
sceUsbdExit();
```

This is a very low level library for sending direct commands to USB devices, so it isn't really ideal to use, but with the help of xerpi, I was able to port one of the `libusb` examples to PS4, and read the raw image of a USB flash drive.

Whilst it may be possible in the future to port a full FAT implementation based on direct USB commands, for now I am just writing my data as the raw image of a USB flash drive using Win32 Disk Imager (similar to `dd` for Linux).

USB findings with kernel access

The PS4 automatically attempts to mount USB flash drives when inserted. Once kernel code execution has been used to enable UART output, the following message is displayed upon insertion of a USB flash drive:

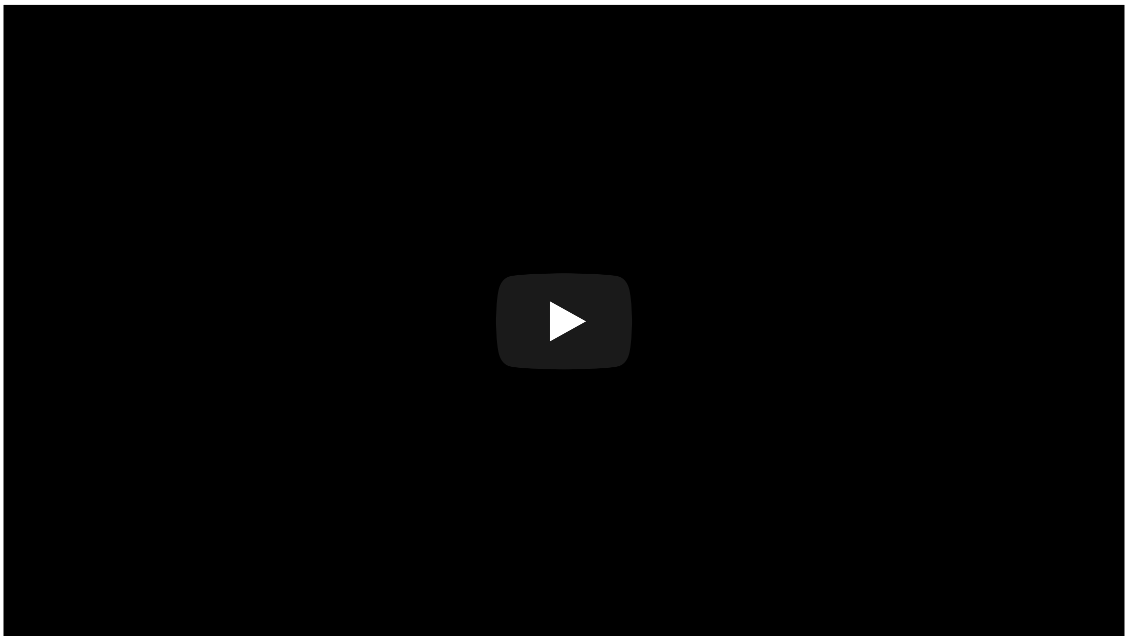
```
ugen0.4: <SanDisk> at usbus0
umass1: <SanDisk Cruzer Edge, class 0/0, rev 2.00/1.26, addr 4> on us
umass1:  SCSI over Bulk-Only; quirks = 0x0000
umass1:2:1:-1: Attached to scbus2
da1 at umass-sim1 bus 1 scbus2 target 0 lun 0
da1: <SanDisk Cruzer Edge 1.26> Removable Direct Access SCSI-5 device
da1: 40.000MB/s transfers
da1: 3819MB (7821312 512 byte sectors: 255H 63S/T 486C)
[SceAutoMount] /mnt/usb0 is now available. fstype=exfatfs, device=/de
MSG AutomounterMelUtil(void sceAutomounterMelUtil::callbackMountAll(v
                device(/dev/da1s1): exfat(mediaType=0x1001) is mounte
```

Only devices formatted as FAT32 will be successfully mounted, and after kernel code execution has been used to escape the filesystem sandbox, they may be accessed from `/mnt/usb0` and `/mnt/usb1`.

However, without a kernel exploit the `libSceUsbd` module remains the only way to access USBs, which actually gives more control over the device, but is less convenient to use for just reading and writing files.

Cinoop

Cinoop is a GameBoy emulator I wrote a while ago. Whilst it isn't one of the best GameBoy emulators out there, I thought it would be a fun project to port to PS4 to show what code execution within the Internet Browser is capable of (using all of the workarounds explained above).



More on processes

Our environment has been restricted such that there are very few ways to interact with other processes meaningfully; I experimented with potential methods of hijacking another process to gain more access but have had little success:

The `fork` (2) system call is disabled, so we can't create new processes.

The `chroot` (61) system call is disabled.

The `libc` function `getprocname` returns an empty string.

The `execve` (59) system call is allowed, and there is also a function called `sceSystemServiceLoadExec` in `libSceSystemService.sprx`, but we have no way of testing either of these since the filesystem is read only and we can't mount USB flash drives. Executable files on the PS4 have a custom header, and the contents are encrypted anyway.

We can copy some of the functions from `libprocstat`, but this functionality is mostly useless since we only have permission to target our own process.

Executable files with kernel access

The following two kernel functions seem to deal with the majority of integrity checks of executable files: `sceSblAuthMgrAuthHeader` and `sceSblAuthMgrIsLoadable`.

With kernel code execution, executable files can be directly decrypted on the console, however there isn't much benefit to this over just loading the module and dumping it from userland.

Root confusion

I mentioned in my last article that `getlogin` returns `"root"`. Whilst the username may be `"root"`, I'm not convinced that it is the conventional `root` that one would expect.

For example, `getuid` should always return 0 for the `root` user, but instead, it returns 1.

I've also demonstrated in my last article that our process is running in a FreeBSD jail, which I'm not sure is possible for a process running as `root`.

I don't understand enough about FreeBSD users and jails to really understand what is going on, but I like to think that Sony somehow named a `non-root` user as `"root"` just to tease us.

Loading modules from their name

We can load modules from their name using `sceKernelLoadStartModule` from `libkernel`:

```
int libPad = sceKernelLoadStartModule("libScePad.sprx", 0, NULL, 0, 0
```

With the module loaded in memory, we can read its base and size, and dump it like before.

This method of loading modules is preferable to the one explained in my last article since it will initialise the imports table, so that you can actually call functions in it, and follow xrefs to other modules like `libc` and `libkernel` in your dump.

This function also lets us dump a few modules that would cause a segmentation fault using the old method.

Finding function offsets from function names

FreeBSD uses system call 337, `kldsym`, to locate the address of a function in a kernel module from its name.

In C, it can be used like this:

```
struct kld_sym_lookup data;
data.version = sizeof(struct kld_sym_lookup);
data.symname = "sys_getpid";

if(kldsym(libKernel, KLDSYM_LOOKUP, &data) == 0) {
    printf("%p\n", data.symvalue);
    printf("%d\n", data.symsize);
}
```

In the PS4 kernel, this function has been disabled, and will always return `0x4e`, `ENOSYS`.

However, Sony implemented a dynamic linker in the PS4 kernel for userland dynamic libraries, and we can use it to resolve userland functions.

System call `591`, `sys_dynlib_dlsym`, has become the basis of the PS4-SDK; once we've loaded a module and got its handle, we can call any functions which we know the name and parameters of.

The following ROP chain will get the offset of the `getpid` wrapper within `libkernel`:

```
var result = chain.data;
var name = chain.data + 8;
```

```
writeString(name, "getpid");
chain.syscall("getFunctionAddressByName", 591, LIBKERNEL, name, resul

chain.execute(function() {
    logAdd(readString(name) + " libkernel offset = 0x" + (getU64from(
});
```

For firmware 1.76, the result is `0xbbb0`.

We can verify this offset from our `libkernel` dump (20 is the `getpid` system call number):

```
000000000000BBB0 getpid          proc near
000000000000BBB0                 mov     rax, 20
000000000000BBB7                 mov     r10, rcx
000000000000BBBA                 syscall
000000000000BBBC                 jb      short loc_BBBF
000000000000BBBE                 retn
000000000000BBBF ; -------------------------------------------------
000000000000BBBF
000000000000BBBF loc_BBBF:
000000000000BBBF                 lea     rcx, sub_DF60
000000000000BBC6                 jmp     rcx
000000000000BBC6 getpid          endp
```

To get other function names to try, you should use the strings view of your disassembler (or just search for `sce` in a hex editor); you'll find that Sony left some useful debug messages in many of the modules.

For example, `libkernel` contains the string `"verify_header: sceKernelPread failed %x\n"`. Now that we've identified a `sceKernelPread` function, we can guess others that may exist, such as `sceKernelPwrite`, and so on.

Unfortunately, `sceKernelPread` and `sceKernelPwrite` aren't very interesting; they are just wrappers for the regular FreeBSD file related system calls.

Since Sony has used a fairly consistent naming convention over the years, you can also try using some PSP function names; many of them also exist in some of the PS4's modules.

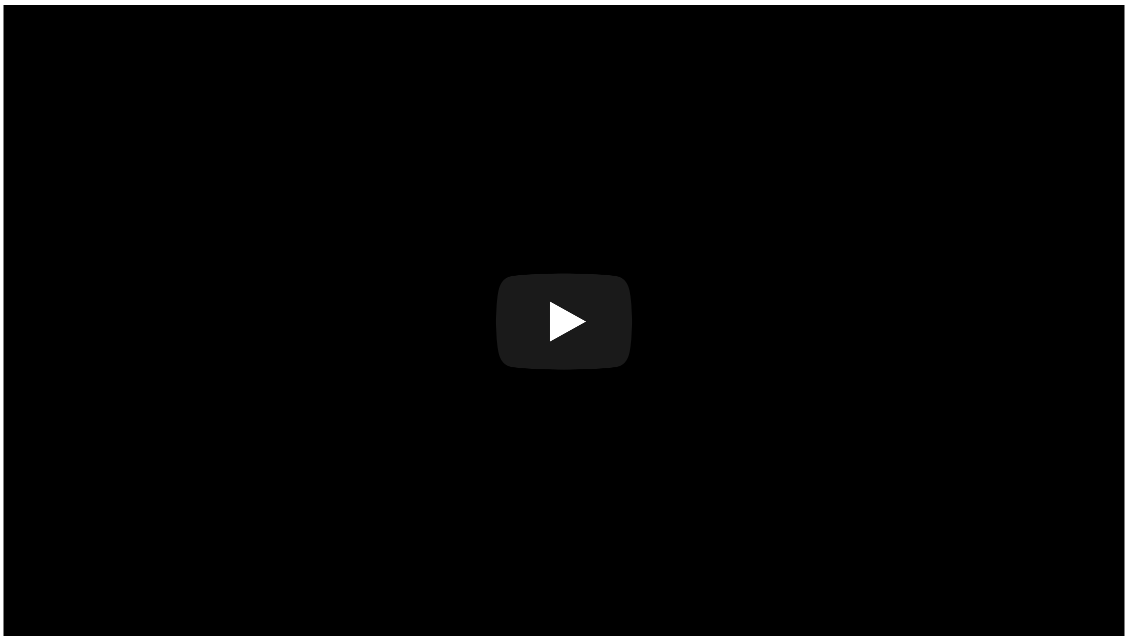
Threads

The `libkernel` module contains an implementation of `libpthread`, but with the Sony naming convention; an example of using threads has been added to the PS4-SDK.

An interesting thing to note is that the threads we create will continue to run in background whilst other applications are active.

To demonstrate this, we can create a thread which will launch the Internet Browser after an arbitrary timeout:

```
int (*sceSystemServiceLaunchWebBrowser)(const char *uri, void *);

void *t(void *n) {
    sceKernelSleep(10);

    sceSystemServiceLaunchWebBrowser("http://google.com/", NULL);

    return NULL;
}

int _main(void) {
    initKernel();

    initLibc();
    initPthread();

    int libSceSystemService;
    loadModule("libSceSystemService.sprx", &libSceSystemService);

    RESOLVE(libSceSystemService, sceSystemServiceLaunchWebBrowser);

    ScePthread thread;
    scePthreadCreate(&thread, NULL, t, NULL, "t");

    return 0;
}
```



Reading memory protection

We can use 2 of Sony's custom system calls, 547 and 572, to read the properties of a memory page (16KB), including its protection:

```
function getStackProtection() {
    var info = chain.data;

    chain.syscall("getMemoryInfo", 547, stack_base, info);

    chain.execute(function() {
        var base = getU64from(info + 0x0);
        var size = getU64from(info + 0x8) - base;
        var protection = getU32from(info + 0x10);

        logAdd("Stack base: 0x" + base.toString(16));
        logAdd("Stack size: 0x" + size.toString(16));
        logAdd("Stack protection: 0x" + protection.toString(16));
    });
}

function getStackName() {
    var info = chain.data;

    chain.syscall("getOtherMemoryInfo", 572, stack_base, 0, info, 0x4

    chain.execute(function() {
        var base = getU64from(info + 0x0);
        var size = getU64from(info + 0x8) - base;
        var name = readString(info + 0x20);

        logAdd("Stack base: 0x" + base.toString(16));
        logAdd("Stack size: 0x" + size.toString(16));
        logAdd("Stack name: " + name);
    });
}
```

The above code shows us that the stack's name is "main stack" and its protection is 3 (read and write).

Listing all memory pages

As you know from my last article, it is difficult to map out all of the PS4's memory due to ASLR (everything is always randomly arranged).

Luckily for us, there is something we can do to partially get around this: if the second argument of system call 572 is set to 1 and we specify an address which isn't mapped, the next mapped memory page will be used.

This means that we can specify any arbitrary address, and always find a valid memory page. For example, specifying 0 as the address will tell us information about the first mapped memory page:

```
var info = chain.data;

chain.syscall("getOtherMemoryInfo", 572, 0, 1, info, 0x40);

chain.execute(function() {
    var base = getU64from(info + 0x0);
    var size = getU64from(info + 0x8) - base;
    var name = readString(info + 0x20);

    logAdd("First page base: 0x" + base.toString(16));
    logAdd("First page size: 0x" + size.toString(16));
    logAdd("First page name: " + name);
});
```

Using this, we can extract a complete list of memory pages accessible from our process:

```
Name                            Address         Size        Protectio
executable                      0x65620000      0x4000      0x5
executable                      0x65624000      0x4000      0x3
anon:000819401c98               0x200578000     0x4000      0x3
anon:00081baf2243               0x20057c000     0x8000      0x3
anon:00081add693a               0x200584000     0x8000      0x3
anon:00081baf22d6               0x20058c000     0x8000      0x3
anon:00081add739e               0x200594000     0x100000    0x3
anon:00081add6ad2               0x200694000     0x8000      0x3
anon:00081add6ad2               0x20069c000     0x8000      0x3
anon:000815405218               0x2006a4000     0x4000      0x3
anon:00081ac4f19e               0x2006a8000     0x8000      0x3
anon:00081add739e               0x2006b0000     0x100000    0x3
anon:00081ba08107               0x2007b0000     0x4000      0x3
anon:00081ad834f7               0x2007b4000     0x4000      0x1
anon:00081add739e               0x2007b8000     0x300000    0x3
stack guard                     0x7ef788000     0x4000      0x0
JavaScriptCore::BlockFree       0x7ef78c000     0x10000     0x3
stack guard                     0x7ef79c000     0x4000      0x0
RscHdlMan:Worker                0x7ef7a0000     0x10000     0x3
stack guard                     0x7ef7b0000     0x4000      0x0
SceWebReceiveQueue              0x7ef7b4000     0x10000     0x3
stack guard                     0x7ef7c4000     0x4000      0x0
SceFastMalloc                   0x7ef7c8000     0x10000     0x3
stack guard                     0x7ef7d8000     0x4000      0x0
```

```
sceVideoCoreServerIFThread      0x7ef7dc000     0x10000     0x3
(NoName)WebProcess.self         0x7ef7ec000     0x4000      0x0
main stack                      0x7ef7f0000     0x200000    0x3
                                0x7ef9f0000     0x4000      0x5
libSceRtc.sprx                  0x802ccc000     0x4000      0x5
libSceRtc.sprx                  0x802cd0000     0x4000      0x3
libSceSystemService.sprx        0x803468000     0x14000     0x5
libSceSystemService.sprx        0x80347c000     0x4000      0x3
libSceSystemService.sprx        0x803480000     0x8000      0x3
libSceSysmodule.sprx            0x8049bc000     0x4000      0x5
libSceSysmodule.sprx            0x8049c0000     0x4000      0x3
libkernel.sprx                  0x808774000     0x34000     0x5
libkernel.sprx                  0x8087a8000     0x2c000     0x3
libSceRegMgr.sprx               0x80a520000     0x4000      0x5
libSceRegMgr.sprx               0x80a524000     0x4000      0x3
libSceSsl.sprx                  0x80d1c0000     0x48000     0x5
libSceSsl.sprx                  0x80d208000     0x8000      0x3
libSceOrbisCompat.sprx          0x80f648000     0x15c000    0x5
libSceOrbisCompat.sprx          0x80f7a4000     0x38000     0x3
libSceOrbisCompat.sprx          0x80f7dc000     0x4000      0x3
libSceLibcInternal.sprx         0x8130dc000     0xd0000     0x5
libSceLibcInternal.sprx         0x8131ac000     0x8000      0x3
libSceLibcInternal.sprx         0x8131b4000     0x18000     0x3
libScePigletv2VSH.sprx          0x815404000     0x74000     0x5
libScePigletv2VSH.sprx          0x815478000     0x2c000     0x3
libSceVideoCoreServerInterface. 0x819400000     0xc000      0x5
libSceVideoCoreServerInterface. 0x81940c000     0x4000      0x3
libSceWebKit2.sprx              0x81ac44000     0x2414000   0x5
libSceWebKit2.sprx              0x81d058000     0x148000    0x3
libSceWebKit2.sprx              0x81d1a0000     0xbc000     0x3
libSceIpmi.sprx                 0x81da60000     0x14000     0x5
libSceIpmi.sprx                 0x81da74000     0x14000     0x3
libSceMbus.sprx                 0x8288a0000     0x8000      0x5
libSceMbus.sprx                 0x8288a8000     0x4000      0x3
libSceCompositeExt.sprx         0x829970000     0x8000      0x5
libSceCompositeExt.sprx         0x829978000     0x44000     0x3
libSceNet.sprx                  0x82ccdc000     0x1c000     0x5
libSceNet.sprx                  0x82ccf8000     0x14000     0x3
libSceNetCtl.sprx               0x833f1c000     0x8000      0x5
libSceNetCtl.sprx               0x833f24000     0x4000      0x3
libScePad.sprx                  0x835958000     0x8000      0x5
libScePad.sprx                  0x835960000     0x8000      0x3
libSceVideoOut.sprx             0x83afe4000     0xc000      0x5
libSceVideoOut.sprx             0x83aff0000     0x4000      0x3
libSceSysCore.sprx              0x83cdf4000     0x8000      0x5
libSceSysCore.sprx              0x83cdfc000     0x4000      0x3
SceLibcInternalHeap             0x880984000     0x10000     0x3
SceKernelPrimaryTcbTls          0x880994000     0x4000      0x3
SceVideoCoreServerInterface     0x880998000     0x4000      0x3
SceLibcInternalHeap             0x88099c000     0xc0000     0x3
SceLibcInternalHeap             0x880a5c000     0x20000     0x3
SceLibcInternalHeap             0x880a7c000     0x490000    0x3
SceLibcInternalHeap             0x880f0c000     0x470000    0x3
anon:00080f64a807               0x912000000     0x100000    0x3
anon:00080f64a98d               0x912100000     0x10000000  0x3
anon:00080f64aaa5               0x922100000     0x4000000   0x5
CompositorClient                0x1100000000    0x200000    0x33
CompositorClient                0x1100200000    0x200000    0x33
CompositorClient                0x1100400000    0x200000    0x33
CompositorClient                0x1100600000    0x200000    0x33
CompositorClient                0x1180000000    0x200000    0x33
CompositorClient                0x1180200000    0x200000    0x33
CompositorClient                0x1180400000    0x200000    0x33
CompositorClient                0x1180600000    0x200000    0x33
CompositorClient                0x1180800000    0x200000    0x33
CompositorClient                0x1180a00000    0x200000    0x33
CompositorClient                0x1180c00000    0x200000    0x33
CompositorClient                0x1180e00000    0x200000    0x33
CompositorClient                0x1181000000    0x200000    0x33
CompositorClient                0x1181200000    0x200000    0x33
CompositorClient                0x1181400000    0x200000    0x33
CompositorClient                0x1181600000    0x200000    0x33
CompositorClient                0x1181800000    0x200000    0x33
CompositorClient                0x1181a00000    0x200000    0x33
CompositorClient                0x1181c00000    0x200000    0x33
CompositorClient                0x1181e00000    0x200000    0x33
CompositorClient                0x1182000000    0x200000    0x33
CompositorClient                0x1184000000    0x200000    0x33
CompositorClient                0x1186000000    0x200000    0x33
CompositorClient                0x1188000000    0x200000    0x33
CompositorClient                0x118a000000    0x200000    0x33
CompositorClient                0x118c000000    0x200000    0x33
CompositorClient                0x118e000000    0x200000    0x33
```

`CompositorClient` is always based at `0x1100000000`, but all other addresses will be different each time.

This list is almost exactly what we expected, a bunch of modules each with their own data and code pages, the stack, some stack guards, and some other miscellaneous mappings.

There is something peculiar though, `CompositorClient` is mapped as `0x33`, which is definitely not a standard FreeBSD memory protection!

GPU

Since the CPU and GPU share a unified memory pool, Sony added their own protection flags to control what the GPU can access as well as keeping the standard FreeBSD protections for the CPU.

These can be found by either reversing the `libSceGnmDriver` module, or just by running some tests and thinking logically:

- CPU Read - 1
- CPU Write - 2
- CPU Execute - 4
- GPU Execute - 8
- GPU Read - 16
- GPU Write - 32

`CompositorClient` is marked as `0x33` (`1 | 2 | 16 | 32`), CPU RW and GPU RW.

Sony handled the GPU protection system very cleverly; we can only give a processor as much access as the other one has, for example:

```
// Give GPU read and write access to stack:
chain.syscall("mprotect", 74, stack_base, 16 * 1024 * 1024, 1 | 2 | 1

// Give GPU read and execute access to WebKit2 module:
chain.syscall("mprotect", 74, module_infos[WEBKIT2].image_base, 16 *
```

But trying to bypass DEP will fail:

```
// Give GPU read and execute access to stack:
chain.syscall("mprotect", 74, stack_base, 16 * 1024 * 1024, 1 | 2 | 1

// Give GPU read and write access to WebKit2 module:
chain.syscall("mprotect", 74, module_infos[WEBKIT2].image_base, 16 *
```

Registry

There is a module called `libSceRegMgr.sprx`, which indicates that Sony added some kind of registry system to the PS4, since FreeBSD doesn't come with one.

All functions in this module are wrappers for system call 532, which was previously thought to be wait6; the first argument is a command.

The fact that `wait6` has been overwritten with a custom Sony system call suggests that the system call numbers are not as similar to standard FreeBSD 9.0 as I initially believed.

Although this module is loaded and used by the Internet Browser, it is restricted from our process; all function calls return `0x80020001`, the Sony equivalent of `EPERM`.

More proof of the lack of kernel ASLR

System call 617 takes at least 1 argument, and returns a kernel pointer; I don't know anything more about this system call, but since the kernel pointer is always the same, we can use it as further evidence that there is no kernel ASLR on firmware 1.76.

Dumping files

Recently, I added a File Browser to PS4-Playground, although I didn't add a way to dump files.

With code execution, files can be dumped very easily. I've added an example to PS4-SDK which shows how to do it.

It is also possible to do using only ROP, but it is a bit more hassle, and must be done in multiple stages.

By using PS4 File Browser, you should be able to find some interesting things to dump; I'll be dumping `/sandboxDir/common/font/DFHEI5-SONY.ttf`.

If the path to the file you want to dump starts with 10 random characters (the sandbox directory), you should note that this path will change each time you reboot the PS4. You can use the ROP chain below to find it:

```
setU64to(chain.data, 11);
chain.syscall("getSandboxDirectory", 602, 0, chain.data + 8, chain.da
chain.write_rax_ToVariable(0);

chain.execute(function() {
    var name = readString(chain.data + 8);
    logAdd(name);
});
```

For me, it was `AaQj0xlzjX`.

For very small files, you can simply read into `chain.data`, but for larger files, you will need to allocate your own memory.

We can do this through the standard `mmap` system call. Refresh the page, and use this chain:

```
chain.syscall("mmap", 477, 0, 0x1000000, 1 | 2, 4096, -1, 0);
chain.write_rax_ToVariable(0);
chain.execute(function() {
    chain.logVariable(0);
});
```

In this example, the address returned was `0x200744000`.

Refresh the page again, and use this chain to read the file and get its size, replace `AaQj0xlzjX` with your sandbox directory and `0x200744000` with whatever address the above chain printed:

```
writeString(chain.data, "/AaQj0xlzjX/common/font/DFHEI5-SONY.ttf");
chain.syscall("open", 5, chain.data, 0, 0);
chain.write_rax_ToVariable(0);
chain.read_rdi_FromVariable(0);
chain.syscall("read", 3, undefined, 0x200744000, 0x1000000);
chain.syscall("fstat", 189, undefined, chain.data);
chain.execute(function() {
    chain.logVariable(0);
    logAdd("Size: " + getU32from(chain.data + 0x48).toString());
});
```

The font I am dumping is 8312744 bytes.

Now open whatever proxy or network tool you use to intercept traffic on your computer. I created a simple C server called TCP-Dump which you can use if you wish.

Refresh the page, and use this chain to send the buffer; replace the IP, port, address, and file size with the appropriate values:

```
sendBuffer("192.168.0.4", 9023, 0x200744000, 8312744);
chain.execute(function() {
    logAdd("Dumped");
});
```

Using cookies, you can pass information to subsequent stages automatically, but I won't go into it now.

You should also note that the filesystem is read only; for example, attempting to overwrite a font will crash your PS4 (but it'll be fine afterwards).

We can also dump the modules located at `/sandboxDir/common/lib/`, but they are encrypted.

Encryption

The most common questions I am asked pertain to encryption. It is a huge part of the PS4's security which prevents us from analysing firmware updates, games, saves and more.

The reason I didn't mention encryption in my last article is because trying to defeat it would be a complete waste of time. The PS4 uses AES (like the PS3 and PS Vita), which is the same type of encryption used by the U.S. government.

People also don't seem to realise that there are multiple encryption keys used within the PS4; even if we found a way to decrypt save data, we still wouldn't be able to decrypt PUP updates for example.

With the current level of access we have to the PS4 there is no way to get any keys: brute forcing them would take longer than the lifetime of the universe even under ideal conditions, and I doubt any of the few engineers at Sony trusted with them would want to lose their job by leaking them.

The only exception to this is would be for implementation mistakes such as the PS3's infamous use of the constant 4 instead of what should have been a random number.

Whilst it is unlikely that Sony has made another mistake like this in the core of the PS4's encryption, it is not uncommon for other companies to accidentally give us access to unencrypted content. If you snoop around various games' update servers, you might find some debug ELFs for example.

Furthermore, encryption on the PS4 is handled by a separate processor, called SAMU, which is very locked down. Even with a kernel exploit, the SAMU processor is one of the few areas which we don't have complete control over. Although we can interact with it to decrypt almost everything, it is impossible to extract any keys so that decryption could be done externally.

Saves

Save data is stored at the following location:

```
/user/home/[userID]/savedata/[titleID]/
```

For example:

```
/user/home/10000000/savedata/CUSA00455/FFXIVSYSTEM.bin
```

We can dump these files, but they are encrypted, and are identical to the files copied from using the PS4's official USB save export feature.

It is unlikely that developers directly deal with this encryption; I assume that the `libSceSaveData` module handles it all.

I was able to load and initialise this module successfully:

```
int libSave = sceKernelLoadStartModule("libSceSaveData.sprx", 0, NULL

int (*sceSaveDataInitialize)(void *);
RESOLVE(libSave, sceSaveDataInitialize);

sceSaveDataInitialize(NULL);
```

But I just received error codes when attempting to mount or read/write save data.

Summary

With the current level of access that code execution has, it is possible to run *some* types of userland homebrew, such as a GameBoy emulator.

However, not being able to use official controllers makes it impractical for standardising any kind of input method; combined with not being able to use the official graphics library, it is clear that homebrew is not yet ready for a full release.

It may not be impossible for our process to read official controllers and to hijack the `libSceVideoOut` module, but it wouldn't be trivial.

`libSceVideoOut` module, but it wouldn't be trivial.

I will continue to run tests in the current environment, and add everything I find to the PS4-SDK, but from what I've seen so far, I don't believe that heavily restricted userland code execution is going to provide a suitable homebrew solution for the masses; a kernel exploit would definitely be the way forward.

Thanks

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