PSP Modules, Exports, Imports and Patches

A short tutorial describing modules used in Sony's PSP

Executive summary

This tutorial describes the use of modules in the PSP. The purpose of this tutorial is two-fold: first, to describe the basic steps to create PSP modules, including pure export modules and applications. Second, to present in detail what happens behind the scenes, how the PSP kernel manages the modules and their import and export functions. Finally, we illustrate how to patch in memory modules dynamically to adapt the behavior of the system to our needs.

The reader is expected to have a basic knowledge of C, the PSPSDK (how to compile and run a Hello World application) and optionally, basic notions of MIPS assembly.

Anissian, February 2007

Anissian.psp@gmail.com

V1.0a2

1 Introduction

So here you are, a starting PSP developer. Most probably, you have already installed the PSPSDK, played around with cygwin -- or installed it in a native POSIX environment -- and you have tried to compile and run the basic SDK samples. You are more or less proficient coding in C, you know how Makefiles work and you have a basic knowledge of assembler (MIPS assembler, but we will touch this briefly, only very basic knowledge such what is a register or a CPU instruction) for the more advanced topics covered.
After reading around and googling a bit, you know the difference between a PRX and an ELF, you know what kernel mode means and what user mode means. You have created your first "Hello world" application, and you can create your own EBOOT.PBP. Now you start to ask "so... what's next?", and two paths lie in front of you: you may join the kernel side of the force, and look at how the inner details of the PSP kernel, or join the user side of the force, considering the PSP and the PSPSDK as a platform with a given set of APIs on top of which develop your own homebrew. In either case, what we explain here we hope it will be useful.

2 Goals and Non-Goals

This tutorial aims at covering the "what next" aspect of PSP programming, narrowing the gap between "I can run a hello world" and "I can create my own firmware or code a downgrader" (the gap remains pretty wide, mind you). Not being a tutorial for starting from scratch, I will not cover the following points, which are already covered in a few, existing tutorials:

- How to set up a PSP development environment with the SDK. Although not strictly necessary, you are supposed to be able to compile your own samples and the code snippets in this tutorial.
- The skeleton of a `"Hello world" main.c` file, and the related `Makefile`.
- How to configure your PSP to run homebrew, either using HENs; SE or OE custom firmwares or similar approaches.
- How to use PSPLINK for launching, transferring or debugging applications.
- Piracy, mod chips or illegal activities.

In this tutorial, we will consider the following topics:

- Very basic MIPS assembly. It is not required, but a basic knowledge of MIPS asm instructions and CPU registers will help you understand a few inner details within this tutorial.
- How to create modules (as PRXs), defining export libraries and export functions (pure export modules). For technical and design reasons (the current PSPSDK does not allow it) only functions will be exported by a given module, not variables. Note that this is not a showstopper, if you want to export a variable, you can always wrap it using accessor and mutator methods (such `as int get_i()` and `void set_i()` for a variable named “`i`”).
- How to create other modules that import exported functions. What is the role of the stubs, how to create import tables, what a `NID` is and how it is computed.
- How to create modules that execute exported functions dynamically.
- How the PSP kernel manages loaded modules. The abstract data types (ADT) involved and how the information of modules is kept in memory.
- How to modify existing modules dynamically ("patching"), and real case scenarios where this has been useful.

For the purposes of this tutorial, a 1.5 firmware is required. Personally, I am using the latest subversion version of PSPSDK on a Linux system and using Firmware 3.03OE-C. For this tutorial we will work on kernel mode both for simplicity and technical reasons. For technical reasons, since we will be looking at the inner details of the module system, and some functions such as `sceKernelFindModuleByName` are kernel-mode only. For simplicity reasons, since in order to overcome these limitations, some kind of voodoo magic is required such as using wrapper functions and mechanisms that allow unconstrained user to kernel calls, named “bridges” such as `kubridge`, or `kvshbridge` which are out of the scope of the tutorial. The resulting `EBOOT.PBP` is supposed to go into the PSP/GAME150 folder and contain a static ELF as the main application. A single directory in the PSP/GAME150 will store the `EBOOT.PBP` and the required PRXes.

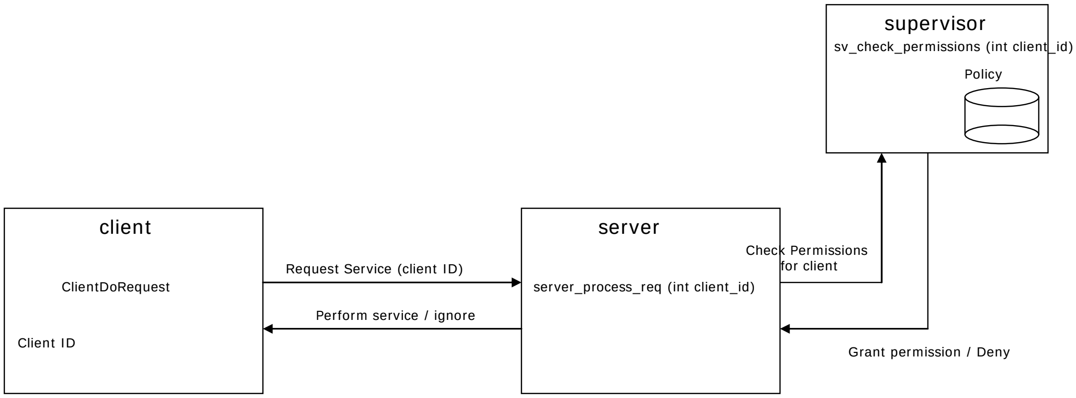
3 Credits

I do not pretend that the ideas presented here are mine or original, some techniques are as old as computer science, and not specific to the PSP. PSP related information has been collected from several sources and sites, in particular:

- PSPDEV site http://www.pspdev.org and lots of forum threads within
- MPH site, notably for the PatchNID functions
- Tyranid excellent PSPLink, the PSPSDK and the PRXTOOL program.
- PspPet ideas and code such as PSARDumper.
- Moonlight / Dark_Alex Custom Firmware Proof of concept.
- The “google guy”
- Many, many more...

4 System Model

For the whole tutorial, we will consider the following system model: an entity, named "client" requests services from a "server" entity, which in turn consults a third entity named "supervisor" which grants privileges to clients based on an identifier. The supervisor is a closed system, and may abstract low level operations.


supervisor
sv_check_permissions (int client_id)
Policy
client
ClientDoRequest
Client ID
Request Service (client ID)
Perform service / ignore
server
server_process_req (int client_id)
Check Permissions for client
Grant permission / Deny

Every system entity is implemented as a module. The Server and Supervisor modules will be export modules.

5 PRX and ELF modules

5.1 What is an ELF

An ELF (Executable and Linking Format) is a format for object files. An ELF is to executables, relocatable and shared objects (modules in general) what a GIF or JPG is to images. The ELF standard basically specifies the binary format of a file. In simple terms, it defines an ELF header followed by a number of sections. Each section has a purpose, such as including the module program instructions, defining what functions the module exports, what functions the module imports; including the initialized

5.2 What is a PRX

A PRX (Relocatable eXecutable) is similar to ELF. (to be completed)

5.3 What is an EBOOT

An EBOOT.PBP file is the equivalent of an “exe” file in Windows. It packs the main executable file and other files that are metadata, such as .png files to be shown at the XMB, a short video for the same purpose and a .SFO file. The format of a PBP file is simple: it contains a fixed-size PBP header, containing: a 4-byte magic number (‘\0’, ‘P’, ‘B’, ‘P’), a u32 version number, and 8 offsets within the PBP file to the packed files: "PARAM.SFO", "ICON0.PNG", "ICON1.PMF", "UNKNOWN.PNG", "PIC1.PNG", "SND0.AT3", "UNKNOWN.PSP" and "UNKNOWN.PSAR".

5.4 Limitations

Using an OE firmware, the 1.50 kernel can start a static elf within an EBOOT without requiring it to be kxploited, either in kernel or user mode. For 3.03, the module needs to be in user mode and in PRX format. Starting from 3.10 OE, the module can be both in static ELF and PRX, as long as it is in user mode. Since the OE firmware is patched so you can load and start modules in kernel mode from a user mode stored in the MS, if you need a kernel mode application, create a user mode wrapper that loads and starts the main application in Kernel memory partition.

6 Initialization

6.1 Module entry points: crt0, start files and module_start

When the PSP kernel loads and starts a module such an application, it reads the EBOOT.PBP file, extracts the ELF or PRX module (which, for the purposes of this tutorial are fundamentally the same thing, barred a few details such as fixed or relocatable memory addresses), parses the module sections, performs some linking to “connect” module imports with already loaded module exports and system syscalls, creates a copy in memory and proceeds to execute the module entry point.

Additionally, the module developer needs to decide whether to have the module entry point executed in a new, separated thread, or to have it executed by the thread that requested the loading and starting of the module. The former means that a new thread, named “main thread” is created (the main name is unrelated to the classical C function also named main), and the latter means that you need to be careful regarding what operations and tasks you perform in the entry point: if your module is supposed to be used as a function library, or simply launch another worker thread, your entry point will be simple, and you may not need a separate thread, but if you need more complex things, you need a main thread (you cannot sleep in your module entry point if you don’t have a main thread). There are macros that define some attributes of your main thread, whether you want it to run in kernel mode, vsh mode, or user mode, whether it will use the VFPU or not, and so on.

For this, the kernel needs basically two things a) to be able to read and `parse module information`, and b) know `where two find the module entry point`. In particular, the kernel expects to find sections in the ELF/PRX giving the module information and a module exported symbol (function), named `module_start`.

Repeating again and again the same tasks and typing the required boilerplate code is an unpleasant task. This is why the SDK abstracts a few details and provides you convenient macros instead.

6.1.1 Module Information

Providing the required module information is made easy using the `PSP_MODULE_INFO` macro:

```
PSP_MODULE_INFO("Module name", 0x1000, 1, 0);
```

This macro is used to state the module name, the module “mode” (kernel, user or vsh) and a major and minor version (v1.0). Behind the scenes, the macro declares a SceModuleInfo that will contain the module attributes, the module name, and pointers to tables containing the module exported (presented for others to use) and imported (used from other modules) symbols, and defines the required ELF/PRX sections where the kernel expects to find this information.

```
/* Module info structure.  Used to declare a module (library or executable).  This
structure is required in all PSP executables. */
typedef struct _scemoduleinfo {
        unsigned short modattribute;
        unsigned char           modversion[2];
        char            modname[27];
        char            terminal;
        void *          gp_value;
        void *          ent_top;
        void *          ent_end;
        void *          stub_top;
        void *          stub_end;
} _sceModuleInfo;

typedef const _sceModuleInfo SceModuleInfo;
```

For the exact details of the macro, check `pspsdk/src/user/pspmoduleinfo.h`

The module mode is where you state whether you want user mode or kernel mode (or others out of the scope such as vsh). Historically, numerical values have been used in most applications. For clarity reasons, you may want to use the more verbose:

```
enum PspModuleInfoAttr
{
        PSP_MODULE_USER    = 0,
        PSP_MODULE_KERNEL = 0x1000,
};
```

6.1.2 Main thread or no main thread?

As discussed earlier, you need to decide whether you want a new thread or not (in case of doubt, yes, you want). If you don’t want / need one, you can use the macro

```
PSP_NO_CREATE_MAIN_THREAD()
```

That will do. On the contrary, if you want the main thread, you may tweak a few parameters using the following macros:

```
PSP_MAIN_THREAD_PRIORITY(priority)

PSP_MAIN_THREAD_STACK_SIZE_KB(size_kb) \

PSP_MAIN_THREAD_ATTR(attr)

PSP_MAIN_THREAD_NAME(s)
```

Or use a single one.

```
#define PSP_MAIN_THREAD_PARAMS(priority, size_kb, attribute) \
        PSP_MAIN_THREAD_PRIORITY(priority); \
        PSP_MAIN_THREAD_STACK_SIZE_KB(size_kb); \
        PSP_MAIN_THREAD_ATTR(attribute)
```

6.1.3 Heap size

Finally, you can define the amount of memory you want to allocate for the heap (the memory pool from where memory is allocated and deallocated by means of malloc / free and similar functions). This is out of the scope of the present tutorial, just to name the following macro:

```
PSP_HEAP_SIZE_KB(size_kb)
```

In some cases, you may want to specify 0 for the heap.

6.1.4 Where is my main?

If you have programmed in C, and browsed the PSPSDK files, you are used to see an application entry point as (a variant of):

```
int main (char* argc, char* argv[])
```

However, we just mentioned that the kernel expects a symbol named `module_start`. What is the relationship between the `module_start` and the `main` functions? The PSPSDK allows you to abstract from these details for most homebrew applications and expect your application to start at the well-known main function (and usually in its own separate thread, but this is not a requirement). This means that you don't need to define a `module_start` since it is done for you. The SDK (`src/startup/crt0.c` and `crt0_prx.c`) define the entry point and call your own main.

Common practice seems to be that if you don't need a main thread, the `module_start` seems enough and no need to define a main function, unless you do complex things. If you want a separate thread, let the sdk (crt0 and startup files) define the thread proc for you, parse the arguments and call your `int main (int argc, char*argv[])` plus a few more things regarding the C library that we will not look into now.

6.2 Pure export module example

For illustrative purposes, let us look at the skeleton of a "pure export" module. A pure export module is a module that acts as a "library" to other modules. The name "pure export" may be a bit misleading, since the name does not imply that the module does not import other symbols. It mostly refers to the fact that it does not "do anything on its own", just a library (similar to .so or .dll files). A pure export module usually declares an empty `module_start` and `module_stop`, it does not need a separate thread for this and does not need to define main. We will later see how to export symbols.

```
#include <pspkernel.h>

PSP_MODULE_INFO("Export Module", 0x1000, 1, 0);
PSP_NO_CREATE_MAIN_THREAD();


int
function_1 ()
{
        return 0;
}

int
module_start (SceSize argc, void* argp)
{
        return 0;
}
```

```
int
module_stop (SceSize args, void *argp)
{
        return 0;
}
```

C code 6-1 basic skeleton of an export module.

6.2.1 Pure export example Makefile

If your module does not need the SDK initialization, add the following line to your Makefile.

```
LDFLAGS += -mno-crt0 -nostartfiles
```

6.3 Another example: extending Custom Firmwares

Another common application of homebrew PRXs is to extend custom firmwares. Some of these firmwares, notably Dark_Alex Open Edition, load and start custom modules when the PSP is started up. Since they extend the capabilities of the firmware, they are commonly referred to as “plugins”. Plugins allow tasks such as taking screenshots, activating USB using few keypresses and more things. The following code shows the skeleton of a kernel mode plugin for Dark Alex custom firmwares, to be used in vsh mode, Basically, it consists of a background thread that creates a file when the L button is pressed. Not very useful, but illustrates the notion of plugins.

Please refer to the README notes of Custom Firmwares regarding how to use and enable them.

```
#include<pspsdk.h>
#include<pspkernel.h>
#include<pspctrl.h>


PSP_MODULE_INFO("psplugin", 0x1000, 1, 0);
PSP_MAIN_THREAD_ATTR(0);

int module_start (SceSize argc, void* argp) __attribute__((alias("plugin_entry")));

/**
 * dump_hello -- creates file plugin_pspain.dat on root dir of MS
 *
 *
 *      Returns -1 on error, 0 otherwise
 */
int plugin_dump_file ()
{
SceUID fd = sceIoOpen ("ms0:/plugin_pspain.dat",
             PSP_O_WRONLY | SP_O_CREAT | PSP_O_TRUNC, 0777);
if (fd < 0)
        return -1;
int a = 1;
int rc;
rc = sceIoWrite (fd, &a, sizeof(int));
if (rc < sizeof(int))
        rc = -1;
else
        rc = 0,
           sceIoClose (fd);
return rc;
}

/**
 * plugin_thread -- worker thread for plugin
 *
 *
 */
int plugin_thread (SceSize argc, void* argp)
{
SceCtrlData pad;
sceCtrlSetSamplingCycle(0);
sceCtrlSetSamplingMode(PSP_CTRL_MODE_DIGITAL);
while (1) {
        sceCtrlPeekBufferPositive (&pad, 1);
        if (pad.Buttons != 0){
                if (pad.Buttons & PSP_CTRL_LTRIGGER){
                        int res = plugin_dump_file ();
                }

                if (pad.Buttons & PSP_CTRL_RTRIGGER){
                        break;
                }
        }
        sceKernelDelayThread(100000);
 }
 sceKernelExitDeleteThread(0);
 return 0;
}
/**
 * plugin_entry -- alias for module_start. Creates plugin worker thread and exits
 *
 */
int plugin_entry (SceSize argc, void* argp)
{
         u32 func = 0x80000000 | (u32)plugin_thread;
         SceUID thread = sceKernelCreateThread("plugin_thread",
             (void*)func, 0x30, 0x10000, 0, NULL);
         if (thread >= 0) {
              sceKernelStartThread (thread, argc, argp);
         }
         return 0;
}
```

7 MIPS architecture and MIPS assembler

7.1 Processor registers

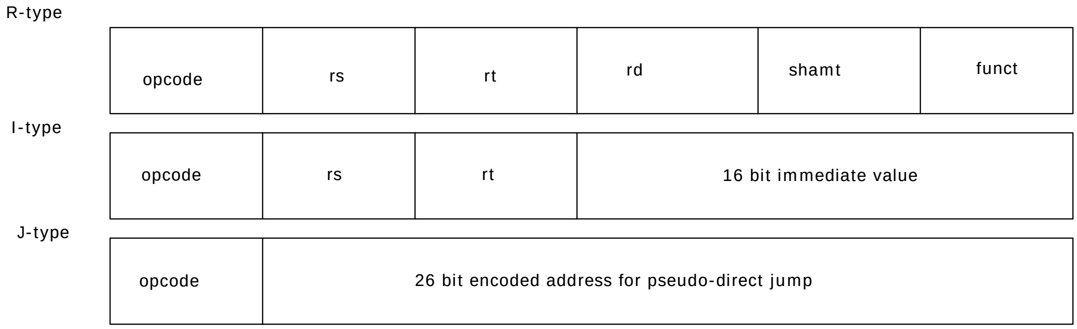
The PSP has a MIPS CPU (Allegrex, a RISC architecture) with 32 registers of 32 bits, numbered from 0 to 31. Each register has a purpose, and the table shows the mapping between register numbers, its mnemonic and a description of its use.

Name	Register	Function
$a0	4	used to pass the first of four arguments
$a1	5	used to pass the second of four arguments
$a2	6	used to pass the third of four arguments
$a3	7	used to pass the fourth of four arguments
$at	1	reserved for the operating system
$fp	30	frame pointer.
$gp	28	global memory pointer
$k0	26	reserved for the operating system
$k1	27	reserved for the operating system
$ra	31	return address
$s0	16	callee-saved registers
$s1	17	callee-saved registers
$s2	18	callee-saved registers
$s3	19	callee-saved registers
$s4	20	callee-saved registers
$s5	21	callee-saved registers
$s6	22	callee-saved registers
$s7	23	callee-saved registers
$sp	29	stack pointer to the first free location
$t0	8	used to hold temporary variables
$t1	9	used to hold temporary variables
$t2	10	used to hold temporary variables
$t3	11	used to hold temporary variables
$t4	12	used to hold temporary variables
$t5	13	used to hold temporary variables

$t6	14	used to hold temporary variables
$t7	15	used to hold temporary variables
$t8	24	used to hold temporary variables
$t9	25	used to hold temporary variables
$v0	2	used to pass values to and from functions
$v1	3	used to pass values to and from functions

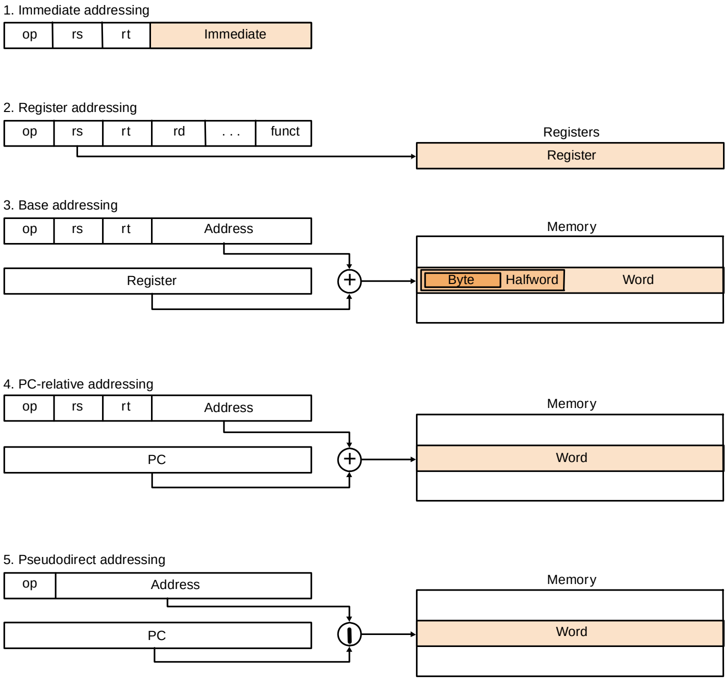
7.2 Basic MIPS instructions

The instruction set has 3 instruction types, a R-type instruction involves registers, an I-type immediate values and a J-type is a pseudo direct address jump. The format of the instructions is:



The following table shows the most common instruction mnemonics. It MIPS assembly, the result of the operation is stored in the first argument. Please note that the mapping between the arguments in the mnemonics and their position in the instruction format does not necessarily match.

CALL	ARGUMENT1	ARGUMENT 2	ARGUMENT3	DESCRIPTION
add	rd	rs	rt	rd = rs + rt (with overflow)
addu	rd	rs	rt	rd = rs + rt (without overflow)
addi	rt	rs	Imm	rt = rs + imm (with overflow)
beq	rs	rt	label	branch to label if (rs==rt)
j	label			jump to label
jal	label			jump to label and save next instruction address in $ra
jr	rs	number		jump to instruction at (rs)
lui	rt	address		upper halfword of rt = 16-bit number
lw	rd	offset(base)		load the word at address into rd
lw	rd			load word at addr offset+base into rd
nop				No operation
or	rd	rs	rt	rd = rs OR rt
ori	rt	rs	Imm	rt = rs OR imm
sw	rt	address		store the word in rt to address
sw	rt	offset(base)		store word in rt to addr offset+base
Syscall				do a system call depending on contents of $v0
xor	rd	rs	rt	rd = rs XOR rt



7.3 Branch targets perform PC relative addressing (close jumps)

7.4 One slot delay jumps

The RISC MIPS processor (and R4000 in particular) has an interesting feature: all jump instructions have a "one slot delay instruction", which means that the instruction just AFTER (in memory) the jump instruction is executed FIRST (in time), and then the jump. From the point of view of the upper layers though, the call is atomic and if the one slot delay instruction raises an exception, it is as if the jump had caused it. For example, when looking at MIPS assembler, it is common to see instructions such as "li $a0, 10000", which in MIPS assembly is a mnemonic for "Load Immediate value 10000 to register argument0", (remember that up to 4 arguments can be passed by using registers, not necessarily on the stack) at a memory address just after a Jump call. The following corresponds to sleep (10000):

```
jal sleep_func
li $a0, 10000
```

7.5 Loading a 32 bit constant: MIPS has only 16 bits of immediate value,

Another particular design feature is that immediate instructions only have 16 bits of immediate value. It is not an issue if the immediate value is small, but when we want to load a 32 bit word into a register, we need to do it in a two step process: first load the most significant 16 bits using “lui” (load upper immediate) and or the result with the lowest 16:

```
  lui $a0, 0xdead
  ori $a0, $a0, 0xbeef
```

7.6 Differences between J and JAL (between jump and call)

In MIPS the j-type instructions are j (opcode 000010) and jal (opcode 000011). The instructions require a 26-bit field to specify the target of the jump. In both cases, the coded address is formed from the bits at positions 27 to 2, and bits 1 and 0 are zero since the instructions are 32-bit aligned (e.g. you can't do JAL 0x03). The full 32-bit jump target address is formed by concatenating the high order 4 bits of the program counter (PC, the address of the instruction following the jump) 26 bits of the target and two 0 bits. For simplicity, just consider that the coded address in the j-type instruction is 26 bits getting the wanted memory address divided by 4.

Both J-type instructions perform a direct call: J means "jump to" and JAL means “Jump and Link”, to be read as "jump to and set the register $ra to the return address where we are supposed to come back"., so the when the PSP executes a jump it does not forget where it was. In other words, the JAL instruction corresponds to a "call" setting the $ra register to the address of the instruction right after the JAL (+4). The called method must then store the $ra register in the stack before calling another method and restoring it to the good value when done so the return address is not clobbered.

The address field of a conditional branch contains a 16 bit two's complement
[(adress of target) - (address of branch) - 4] / 4

In order to better understand the differences, consider this pseudo-code snippet:

```
void bar ()
{
 ...
}

void foo ()
{
 bar ();
}

int main ()
{
 foo ();
}
```

A simplified equivalent assembly code would look like this:

```
main:
     0x10000004    jal foo
     0x10000008    nop
     0x1000000C    ...
     0x10000010    ...
```

At this moment, the register `$ra` points at the address `0x1000000C` (we do not consider the one-slot delay instruction here)

```
foo:
     0x20000000    addiu     $sp, $sp, -4  ;reserve 4 bytes in the stack
     0x20000004    sw        $ra, 0($sp)   ;store the ra in the stack
     0x20000008    jal       bar           ;call bar with $ra = 0x2000000C
     0x2000000C    nop
     0x20000010    lw        $ra, 0($sp)
     0x20000014    jr        $ra           ;restore sp, jump to return address
     0x20000018    addiu     $sp, $sp, 4
```

When the instruction `jal bar` is executed, the register `$ra` points at `0x20000010`

```
bar:
     0x30000000    addiu     $sp, $sp, -4  ;reserve 4 bytes in the stack
     0x30000004    sw        $ra, 0($sp)   ;store the ra in the stack
 ...
                   lw        $ra, 0($sp)
                   jr        $ra           ;restore the sp and jump to return address
                   addiu     $sp, $sp, 4
```

When the bar function ends, the control is transferred to address `0x200000010`, then `foo` finishes and returns to main (`0x1000000C`).

So, if we replaced the "jal bar" by "j bar" what would happen? Since the $ra is not set, it still points to 0x1000000C. The bar function would indeed save the value in stack and set it back when done.... to 0x1000000C. Indeed, the jr $ra at the end off bar would return control....to main! (Leaving foo unfinished, plus side effects such as stack leaks and sp shifts).... bad things would happen.

8 The Client / Server / Supervisor example

Now that we have the basics covered, let's proceed with our example.

8.1 The Supervisor Module

The first module is the Supervisor Module. It is a kernel mode pure export module, which exports a single function. A single `supervisor_main.c` file is enough:

```
#include <pspkernel.h>

PSP_MODULE_INFO("PatchMod_Supervisor", 0x1000, 1, 0);
PSP_NO_CREATE_MAIN_THREAD();
```

```
const char*
sv_check_permissions (int client)
{
        const char* p;

        sceKernelDelayThread (10000);
        if (client == 123)
                p = "yes";
        else
                p = "no";

        return p;
}



int
module_start (SceSize argc, void* argp)
{
        return 0;
}


int
module_stop (SceSize args, void *argp)
{
        return 0;
}
```

C code 8-1 source code for the supervisor module

8.1.1 Module Info and Main Thread

As you know, it is a kernel mode module, named PatchMod_Supervisor. The module name is used in several parts, and is stored in memory once the module is loaded. The use of the macro PSP_NO_CREATE_MAIN_THREAD states that the main thread should not be created thus module_start is run from the caller thread.

8.1.2 The exports file

In order to export a set of libraries and functions, you need to define an exports file. The following text needs to be included in the supervisor_exp.exp file. The meaning of the macros will be given shortly.

```
PSP_BEGIN_EXPORTS

PSP_EXPORT_START(syslib, 0, 0x8000)
PSP_EXPORT_FUNC(module_start)
PSP_EXPORT_VAR(module_info)
PSP_EXPORT_END

PSP_EXPORT_START(supervisor, 0, 0x0001)
PSP_EXPORT_FUNC(sv_check_permissions)
PSP_EXPORT_END

PSP_END_EXPORTS
```

C code 8-2 exports file for the supervisor module.

What does this mean? This means that this module is exporting 2 libraries, a (compulsory) library named "syslib" that exports module_start and module_info and another library named "supervisor" that exports the function "sv_check_permissions". The macros used are straightforward: PSP_BEGIN_EXPORTS / PSP_END_EXPORTS to delimit the exports, PSP_EXPORT_START to start the exports for a given library (library name, library version, flags), PSP_EXPORT_FUNC to export a function and finally PSP_EXPORT_VAR to export a variable (although Tyranid says “don’t do it” :o).

Special mention to the "flags" part: 0x8000 is reserved for the syslib library; 0x0001 means that if used, the library uses "direct jump" e.g. a user module importing functions from another user module, or a kernel module importing functions from a kernel module, and this is what will be used. Finally, if the flags are 0x4001, the syscall mechanism is used (for example, when exporting kernel functions to be used by user modules). A syscall is somewhat of a "safe way" by which a user mode module can access the inner parts of the system. In more complex systems, the syscall wrapper performs extra checks regarding the memory address spaces, the nature of the memory, the call, and may validate or map call arguments, used in general, when a kernel module wants to provide some functionality to user mode modules. In our case, since we are working in kernel mode, we will be using direct jumps (J opcodes in assembly).

The corresponding `Makefile` is as follows. As you know, we ask the toolchain to build a PRX (`supervisor.prx`), and specified in `supervisor_exp.exp` the list of exports. This module will export a single library "supervisor" and a single function. The psp-build-exports is required to build the module exports (using `psp-build-exports -k supervisor_exp.exp` will generate a temporary `supervisor_exp.c` which will be compiled, linked and removed, this is automated by the SDK build system) and to generate the module stubs or import table (using `psp-build-exports -s -k -v supervisor_exp.exp` which will generate `supervisor.S` for easy inclusion in other modules that may use the exported functions). You may also use prxtool (installed separatedly, not in the SDK) using prxtool -u -k supervisor.prx, but you will not get the literal name. Being a pure export module, it declares no "main" and we do not link the crt0 and other start files.

```
release: all
 psp-build-exports -k supervisor_exp.exp
 psp-build-exports -s -k -v supervisor_exp.exp

PSPSDK = $(shell psp-config --pspsdk-path)
PSPLIBSDIR = $(PSPSDK)/..
USE_KERNEL_LIBS=1
TARGET = supervisor
OBJS   = supervisor_main.o

CFLAGS = -O2 -G0 -Wall
ASFLAGS = $(CFLAGS)
LDFLAGS += -mno-crt0 -nostartfiles

BUILD_PRX=1
PRX_EXPORTS=supervisor_exp.exp

include $(PSPSDK)/lib/build.mak
```

C code 8-3 Makefile for the supervisor module

Put these 3 files (`Makefile,` `supervisor_exp.exp` and `supervisor_main.c`) in a folder and compile the module. If everything goes well, you will have a `supervisor.prx` and a `supervisor.S`. The `supervisor.S` is commonly known as the "stub" file and in compiled form it will allow other modules to use the exported function easily, if they link the resulting .o files. This file can be generated from the exp file using "`psp-build-exports -s -k -v supervisor_exp.exp`" or the PRX directly (although using the latter the literal name of the function is lost).

8.1.3 Functions and NIDs

If you look at the stub file, you will see references to symbols of the form library_XXXXXXXX. Where does this naming scheme come from? Well, in the PSP, functions in a module are named by concatenating the library they are exported (supervisor) and a 4-byte identifier called the NID. The NID is obtained literally by computing the SHA-1 of the function name.

XXX add Sha-1 examples and PSP code.

Normally NIDs must be unique at least within a module / library context (if you ever happen to have 2 functions in the same module, same library and which happen to have the same NID, congratulations, you are a extremely lucky developer. That said, change the name of one of them to avoid subsequent headaches, believe me).

When reverse engineering modules, mapping NIDs to literal function names and obtaining meaningful prototypes is very useful (cfr. Nidattack by djhuevo et al.)

8.1.4 Assembler dump of the server.prx module

Since it is a small module, let us inspect the object file code (in assembler form). For this we can use

```
prxtool -w supervisor.prx
```

Let us examine the output assembler for our supervisor module and related sections. As mentioned before, information about the module is stored in the `.rodata.sceModuleInfo` section[1]

```
; ==== Section .rodata.sceModuleInfo - Address 0x000000A0 Size 0x00000040 Flags 0x0002
           - 00 01 02 03 | 04 05 06 07 | 08 09 0A 0B | 0C 0D 0E 0F - 0123456789ABCDEF
--------------------------------------------------------------------------------------
0x000000A0 - 00 10 00 01 | 50 61 74 63 | 68 4D 6F 64 | 5F 53 75 70 - ....PatchMod_Sup
0x000000B0 - 65 72 76 69 | 73 6F 72 00 | 00 00 00 00 | 00 00 00 00 - ervisor.........
0x000000C0 - 20 82 00 00 | 60 00 00 00 | 80 00 00 00 | 88 00 00 00 -  ...`..........
0x000000D0 - 9C 00 00 00 | 00 00 00 00 | 00 00 00 00 | 00 00 00 00 - ................
```

In an ELF or PRX module, the body of the program is kept in the `.text` section, as follows. In particular, the method "`const char* sv_check_permissions (int client)`"

```
; ==== Section .text - Address 0x00000000 Size 0x00000054 Flags 0x0006

; ======================================================
; Subroutine supervisor_B2C390CB - Address 0x00000000
; Exported in supervisor
supervisor_B2C390CB:            ; Refs: 0x000000E4
        0x00000000: 0x27BDFFF8 '...'' - addiu      $sp, $sp, -8
        0x00000004: 0xAFB00000 '....' - sw         $s0, 0($sp)
        0x00000008: 0x00808021 '!...' - move       $s0, $a0
        0x0000000C: 0xAFBF0004 '....' - sw         $ra, 4($sp)
        0x00000010: 0x0C000015 '....' - jal        ThreadManForKernel_CEADEB47
        0x00000014: 0x24042710 '.'.$' - li         $a0, 10000
        0x00000018: 0x3C030000 '...<' - lui        $v1, 0x0
        0x0000001C: 0x2402007B '{..$' - li         $v0, 123
        0x00000020: 0x12020003 '....' - beq        $s0, $v0, loc_00000030
; Data ref 0x00000114 ... 0x00736579 0x00006F6E 0x65707573 0x73697672
        0x00000024: 0x24630114 '..c$' - addiu      $v1, $v1, 276
        0x00000028: 0x3C020000 '...<' - lui        $v0, 0x0
; Data ref 0x00000118 ... 0x00006F6E 0x65707573 0x73697672 0x0000726F
        0x0000002C: 0x24430118 '..C$' - addiu      $v1, $v0, 280

loc_00000030:           ; Refs: 0x00000020
        0x00000030: 0x8FBF0004 '....' - lw         $ra, 4($sp)
        0x00000034: 0x8FB00000 '....' - lw         $s0, 0($sp)
        0x00000038: 0x00601021 '!.`.' - move       $v0, $v1
        0x0000003C: 0x03E00008 '....' - jr         $ra
        0x00000040: 0x27BD0008 '...'' - addiu      $sp, $sp, 8

; ======================================================
; Subroutine module_start - Address 0x00000044
; Exported in syslib
module_start:           ; Refs: 0x000000F0
        0x00000044: 0x03E00008 '....' - jr         $ra
        0x00000048: 0x00001021 '!...' - move       $v0, $zr
        0x0000004C: 0x03E00008 '....' - jr         $ra
        0x00000050: 0x00001021 '!...' - move       $v0, $zr
```

As shown, we can see that there are 2 subroutines, `supervisor_B2C390CB` and `module_start`, exported in the libraries supervisor and syslib respectively. One may suspect that the `supervisor_B2C390CB` corresponds to our `sv_check_permissions` function, and that `ThredManForKernel_CEADEB47` to the `sceKernelDelayThread`.

[1] It is worth mentioning the relation between this module section and the PSP_MODULE_INFO macro used in the source code. See the PSPSDK initialization files for details.

Finally, `module_start` is really simple: return to the return address returning zero (note the use of the constant value register $zr). I must admit I don’t know why the jr and move instructions appear twice, though.

8.1.5 The method sv_check_permissions in detail

A closer look at the assembly can give us a better understanding of how it works:

```
addiu $sp, $sp, -8
```

Substract 8 (bytes) from the stack pointer, thus leaving room for 2 32-bit words in the stack. The stack "grows up", that is, leaving room for local variables, local copies of registers and local copies of information such as the return address makes the sp (stack pointer) register move up in memory (lower memory addresses).

```
sw $s0, 0($sp)
```

Store the contents of register s0 (temporary “all purpose register” that needs to be saved by the callee. In other words, the caller expects this register not to change if a function is called. If the callee modifies the register, it needs to be left as it was before returning to the caller.

```
move $s0, $a0
```

Store the argument of the function (client Id) in register s0

```
sw $ra, 4($sp)
```

Store the contents of the return address in the stack (since we will be calling ThreadManForKernel_CEADEB47) the contents of the register will change.

```
jal ThreadManForKernel_CEADEB47
li  $a0, 10000
```

Call the method with one-slot delay jal, storing the numerical immediate value 10000 as first (and sole) argument of the method, using argument register $a0.

```
li $v0, 123
```

Store the numerical immediate value 123 in register v0

```
beq $s0, $v0, loc_00000030
```

Compare register $s0 (argument client id) with register $v0 (numerical 123) and "branch if equal" to location loc_00000030, after having stored the value 276 (0x114) in v1. The 276 corresponds to the relative address of the "yes" string. Likewise, 280 corresponds to the relative address of the "no" string. So depending on the comparison we have addiu $v1, $v1, 276 or lui $v0, 0x0 - addiu $v1, $v0, 280 thus storing in v1 the address of the string we want to return. Note that the compiler has removed the 'p' local variable, optimized away by using of registers.

```
lw $ra, 4($sp)
```

Get the original return address (from the stack, M[sp + 4]) and store it in the $ra register.

```
lw $s0, 0($sp)
```

Leave the register s0 as it was when our method was called.

```
move $v0, $v1
```

Move the result value (a pointer to the static string) in register v0, where the return value is stored.

```
jr   $ra
addiu $sp, $sp, 8
```

One-slot delay jump to the address in register $ra (return address) after having left the stack as it was expected by the caller.

Using the same tool, we can inspect the other PRX sections and realize a few things: that the `sceKernelDelayThread` corresponds to the `ThreadManForKernel_CEADEB47` identifier, and imported from an external, module.

```
; ==== Section .sceStub.text - Address 0x00000054 Size 0x00000008 Flags 0x0006

; ======================================================
; Subroutine ThreadManForKernel_CEADEB47 - Address 0x00000054
; Imported from ThreadManForKernel
ThreadManForKernel_CEADEB47:          ; Refs: 0x00000098 0x00000010
 0x00000054: 0x03E00008 '....' - jr         $ra
 0x00000058: 0x00000000 '....' - nop
```

As detailed later, this is a Stub: a placeholder that will be changed when the module is loaded, once the address of the `ThreadManForKernel_CEADEB47` is resolved, the module will be linked and the now-empty jr $ra (which simply returns to the caller address) will be changed by a jump to the function.

String literals used in the program (const char*) are stored in the `.rodata` section. Note that they are zero-ended and 32-bit aligned. The `“yes”` is at relative address `0x0114` and the `“no”` is at `0x0118`.

```
; ==== Section .rodata - Address 0x00000114 Size 0x00000013 Flags 0x0032
           - 00 01 02 03 | 04 05 06 07 | 08 09 0A 0B | 0C 0D 0E 0F - 0123456789ABCDEF
-------------------------------------------------------------------------------------
0x00000114 - 79 65 73 00 | 6E 6F 00 00 | 73 75 70 65 | 72 76 69 73 - yes.no..supervis
0x00000124 - 6F 72 00 -- | -- -- -- -- | -- -- -- -- | -- -- -- -- - or.............
```

8.2 The Server Module

The server module services client requests, by exporting a method named `server_process_req`. When a client requests a service, the server module checks whether the supervisor agrees or not. Usually, the server performs some tasks if the client has been granted privileges. For the purposes of this tutorial, the module is kept simple. If a client is granted privileges, the function returns 1 (true) or 0 (false) otherwise. A client will be granted privileges if its identifier is “123”.

The module is also a pure export PRX module in Kernel mode, so a lot of things discussed for the supervisor module apply. The server module not only exports a method, `server_process_req`, but also imports the Server exported `sv_check_permissions`. The prototype of the latter is given in the source code. More complex and real systems, the developer who does not want to distribute source code will provide you with the PRX, the stub file .S and a header file with function prototypes for inclusion and linking. In fact, this is what is being done by reverse engineering in a few kernel and user modules in PSP firmwares, to alleviate developers of the boilerplate code and fastidious tasks of "hand" linking.

8.2.1 The server module source file

The C source for the server module is given below

```
#include <pspsdk.h>
#include <stdlib.h>
#include <string.h>
#include <pspkernel.h>

PSP_MODULE_INFO("PatchMod_Server", 0x1000, 1, 0);
PSP_NO_CREATE_MAIN_THREAD();

const char* sv_check_permissions (int client_id);




/**
 * Processes a client request
 *
 * @param client_id – client identifier. This is checked with the supervisor
 *
 * @returns 1 if request was processed successfully.
 */

int
server_process_req (int client_id)
{
        sceKernelDelayThread (10000);
        const char* super_string = sv_check_permissions (client_id);
        if (super_string[0] == 'y')
                return 1;
        else
               return 0;
}


int
module_start (SceSize argc, void* argp)
{
        return 0;
}

int
module_stop (SceSize args, void *argp)
{
        return 0;
}
```

C code 8-4 source for the server module.

8.2.2 The server module import file

Lets see how automatic importing works. We saw how it is possible to generate a stub file when compiling a module that exports symbols. This file (whose name usually ends in .S) is copied to an appropriate directory, compiled and linked to the module that imports the symbols. The stub (also name import table) file is as follows (this format is known as the new format, which has advantages with regard to existing practice):

```
 .set noreorder

#include "pspimport.s"

// Build files
// supervisor_0000.o supervisor_0001.o

#ifdef F_supervisor_0000
 IMPORT_START "supervisor",0x00090000
#endif
#ifdef F_supervisor_0001
 IMPORT_FUNC  "supervisor",0xB2C390CB,sv_check_permissions
#endif
```

This assembly source file will, in fact, be compiled several times, each time with a given define (-D flag) allowing for ulterior optimizations. A Makefile can be used to automate this.

8.2.3 The server module Makefile

```
release: all

PSPSDK = $(shell psp-config --pspsdk-path)

PSPLIBSDIR = $(PSPSDK)/..

TARGET = server
OBJS   = server_main.o $(SUPERVISOR_OBJS)

CFLAGS = -O2 -G0 -Wall
ASFLAGS = $(CFLAGS)

LDFLAGS += -mno-crt0 -nostartfiles
USE_KERNEL_LIBS=1

BUILD_PRX=1

PRX_EXPORTS=server_exp.exp

SUPERVISOR_OBJS = supervisor_0000.o supervisor_0001.o

$(SUPERVISOR_OBJS): supervisor.S
            psp-gcc $(CFLAGS) -DF_$* $< -c -o $@


include $(PSPSDK)/lib/build.mak
```

8.3 The Client Application

8.3.1 File Structure

Create a Project directory and a corresponding supervisor and server subdirs, so the whole tutorial can be compiled. The Makefile for the client module is as follows.

```
SUBDIRS = supervisor server
release:
 @for i in $(SUBDIRS); do echo "make all in $$i...";  (cd $$i; $(MAKE)  all; cp $$i.prx ..);
done
 make all
```

```
allclean:
 make clean
 @for i in $(SUBDIRS); do echo "Clearing in $$i..."; (cd $$i; $(MAKE) clean; rm -rf $$i.prx); done


PSPSDK = $(shell psp-config --pspsdk-path)

PSPLIBSDIR = $(PSPSDK)/..

TARGET = client
OBJS   = main.o
CFLAGS = -O2 -G0 -Wall
ASFLAGS = $(CFLAGS)

EXTRA_TARGETS = EBOOT.PBP
PSP_EBOOT_TITLE = ModPatch Client

include $(PSPSDK)/lib/build.mak

install:
 mkdir -p /media/disk/psp/game150/modpatch
 cp EBOOT.PBP server.prx supervisor.prx /media/disk/psp/game150/modpatch
 sync
```

8.3.2 Boilerplate code

Other than the boilerplate code that you already know, when developing homebrew applications, it is often required to call a few kernel mode functions that ease the pain. The original PSP system has several limitations regarding modules such as which device the modules can be loaded from, whether they are signed or encrypted and other similar restrictions (such as loading a module in kernel memory partition from a user module, etc.). Later, we will see some of the techniques used for such purposes, but for the time being, just consider that (in kernel mode) you need to add the following lines (usually put in `main` or `module_start` if in kernel mode, or in the application "`constructor`", see PspSDK doc, if the application is in user mode).

```
 pspSdkInstallNoDeviceCheckPatch();
 pspSdkInstallNoPlainModuleCheckPatch();
 pspSdkInstallKernelLoadModulePatch();
```

8.3.3 Preprocessor directives and macros

A few macros are defined to simplify the dynamic construction of MIPS instructions:

```
typedef int (*METHOD) (int client_id);


// null the 6 MSB bits (the opcode bits)
// and divide x by 4 to get the J-Instr parameter.
#define MIPS_J_ADDRESS(x) (((u32)((x)) & 0x3FFFFFFF) >> 2)

#define NOP       (0x00000000)
#define JAL_TO(x) (0x0E000000 | MIPS_J_ADDRESS(x))
#define J_TO(x)   (0x08000000 | MIPS_J_ADDRESS(x))
#define LUI(x,y)  (0x3C000000 | ((x & 0x1F) << 0x10) | (y & 0xFFFF))
```

```
#define PSPDEBUG

#define printf pspDebugScreenPrintf
#if defined(PSPDEBUG)
#define     dbg_log   pspDebugScreenPrintf
#else
#define     dbg_log(...)
#endif
```

8.3.4 Typedefs

For clarity purposes, we use the following typedefs:

- PspNID is used to store a PSP NID, an unsigned 32 bit word
- MIPSInstruction is used to store a MIPS Instruction

- MemAddress is a 32-bit aligned memory address. Dereferencing a pointer gives you the 32 bit word found at that address. As per std. C pointer arithmetic, incrementing a MemAddress increases the absolute numeric address in 4.

```
typedef u32   PspNID;
typedef u32*   MemAddress; // a 32-bit aligned memory address (4-byte aligned arithmetic
0x0000 -> 0x0004)
typedef u32   MIPSInstruction;
```

For example, the following function can be used to dump the memory region corresponding to the text section to the screen:

```
/**
 * Dumps a memory region given by start and end on debug console
 *
 * @param start - starting memory address (32 bit aligned, u32*)
 * @param end - ending memory address (32 bit aligned, u32*)
 *
 */

static void
pspDumpMemRegion (MemAddress start, MemAddress end)
{
 MemAddress a = start;
 MIPSInstruction I;
 while (a <= end) {
        I = (MIPSInstruction) (*a);
        printf("Instr: 0x%08X: 0x%08X\n", (u32)a, (u32)I);
        a++;
 }
}
```

C code 8-5 pspDumpMemRegion

8.3.5 Capturing the “Press Home to quit”

As you may have seen in the SDK samples, it is common practice to add some code at startup so your program or application, despite of what it is doing at a given moment, can be cancelled by pressing “home”. For this, a possible approach is that your thread would continually read or peek the control status for the “home” key. This solution, which can only be done in kernel mode, anyway, may not be convenient. A safer, cleaner approach required if, for example, your thread is sleeping or actively doing anything, is the following: the kernel allows you to create and register a callback automatically called at exit. For this, you create a user mode worker thread for this purpose: the thread procedure simply creates and registers a callback for exiting the application and goes to sleeps (but waiting for callbacks, `sceKernelSleepThreadCB`). When the user presses home, the kernel pops a dialog asking for confirmation, and if the user replies “yes”, your callback method of the worker thread is called, allowing you to quit your application by calling `sceKernelExitGame.` The drawback is that you have created a thread just for this. In most cases, just add some code as shown below, and call `AppSetupExitCallback` at the beginning.

```
/**
 * Called by Kernel when user quits the application.
 *
 *
 *  @returns 0
 */
static int
_app_exit_callback(int arg1, int arg2, void *common)
{
 sceKernelExitGame();
 return 0;
}

static int
_app_ExitCallbackThreadProc (SceSize args, void *argp)
{
 int cbid = sceKernelCreateCallback("Exit Callback", _app_exit_callback, NULL);
 sceKernelRegisterExitCallback(cbid);
 sceKernelSleepThreadCB();
 return 0;
}


static int
AppSetupExitCallback (void)
{
 int thid = sceKernelCreateThread("exit_callback_thread",
        _app_ExitCallbackThreadProc, 0x11, 0xFA0, THREAD_ATTR_USER, 0);
 if (thid >= 0)
 {
        sceKernelStartThread(thid, 0, 0);
 }
 return thid;
}
```

C code 8-6 boilerplate code to setup a thread that manages the exit callback

8.3.6 Loading the modules

After doing some initialization (including preparing the screen and setting up the exit callback), the application loads and starts both PRX modules, using the `pspModuleLoadStartInKernelPart:`

```
int
main (int argc, char* argv[])
{
   (void)argc;
   (void)argv;
 pspDebugScreenInit();
 pspSdkInstallNoDeviceCheckPatch();
 pspSdkInstallNoPlainModuleCheckPatch();
 pspSdkInstallKernelLoadModulePatch();
 AppSetupExitCallback();

 SceModule* modsuper  = pspModuleLoadStartInKernelPart ("supervisor.prx");
 SceModule* modserver = pspModuleLoadStartInKernelPart ("server.prx");
```

The method `pspModuleLoadStartInKernelPart` is very similar to those present in the SDK, and basically loads the module in a Kernel memory partition[2]

```
/**
 * Loads and Starts module in Kernel memory partition
 *
 * @param modpath - path to the module to load (cfr. sceKernelLoadModule)
 *
 * @return (SceModule*) address.
 * @returns the address of the module struct for the loaded module
 *          or NULL if either sceKernelLoadModule or sceKernelStartModule
 *          failed.
 */

static SceModule*
pspModuleLoadStartInKernelPart (const char* modpath)
{
 SceKernelLMOption option = {
         .size     = sizeof(option),
         .mpidtext = (SceUID) 1, // SceUID mpid = 1 kernel partition
         .mpiddata = (SceUID) 1,
         .position = 0,
         .access   = 1
 };
```

[2] Let us note that with custom firmwares it may be possible to load modules in a kernel partition from a user thread. This is a means to overcome some limitations e.g. in the 3.X OE firmwares, since they require a user mode PRX or static ELF to be in the application EBOOT.PBP. If your application needs to be in kernel mode, use bootstrapping: use a user mode thread that loads a module in kernel memory and starts it. The loaded module will be in kernel mode (provided it has been compiled as such).

```
SceUID modid = sceKernelLoadModule(modpath, 0, &option);
if (modid < 0) {
        dbg_log ("sceKernelLoadModule('%s') "
                            "failed with %x\n", modpath, modid);
        return NULL;
}

int status = 0;
int res = sceKernelStartModule(modid, 0, NULL, &status, NULL);
(void)status;
if (res < 0) {
        dbg_log ("sceKernelStartModule('%s') "
                "failed with %x\n", modpath, res);
        sceKernelUnloadModule(modid);
        return NULL;
}
return sceKernelFindModuleByUID(modid);
}
```

C code 8-7 pspModuleLoadStartInKernelPart

8.3.7 Managing modules

The kernel maintains a list of loaded modules and their related information. Most operations requiring access to the involved structs require kernel mode, since user mode applications are limited to a few module tasks. For example, the following snippet outputs the module name if the module name does not start with sce (a dirty hack to detect homebrew modules).

```
/**
 * Outputs the name of the module on DBG console.
 *
 * @param mod - module.
 *
 * @return (void)
 */

static void
print_module (SceModule* mod)
{
if (!mod)
        return;

if (strncmp (mod->modname,"sce", 3) != 0) {
        dbg_log ("mod: %s\n", mod->modname);
}
}
```

C code 8-8 print_module function

In order to get a list of the loaded modules (each one represented by a `SceModule`*), you can use the `sceKernelGetModuleIdList` fuction, which basically fills a passed array of `SceUIDs` with the uids of the loaded modules. You may then use the function `sceKernelFindModuleByUID` to map that module uid to the corresponding `SceModule`. For convenience reasons, it is good to have a function that iterates the list of loaded modules, applying a given function as argument to each loaded module:

```
/**
 * Foreach iteration for the list of Loaded Modules
 *
 * @param func function pointer to callback for each element
 *
 * @return (void)
 */

static void
pspForeachLoadedModule (void (*func) (SceModule*))
{
SceUID ids[512];
memset (ids, 0, 512 * sizeof(SceUID));
int count = 0;
sceKernelGetModuleIdList(ids, 512 * sizeof(SceUID), &count);

int p;
for (p = 0; p < count; p++)
{
        SceModule* mod = sceKernelFindModuleByUID(ids[p]);
        if (mod)
                (*func)(mod);
}
}
```

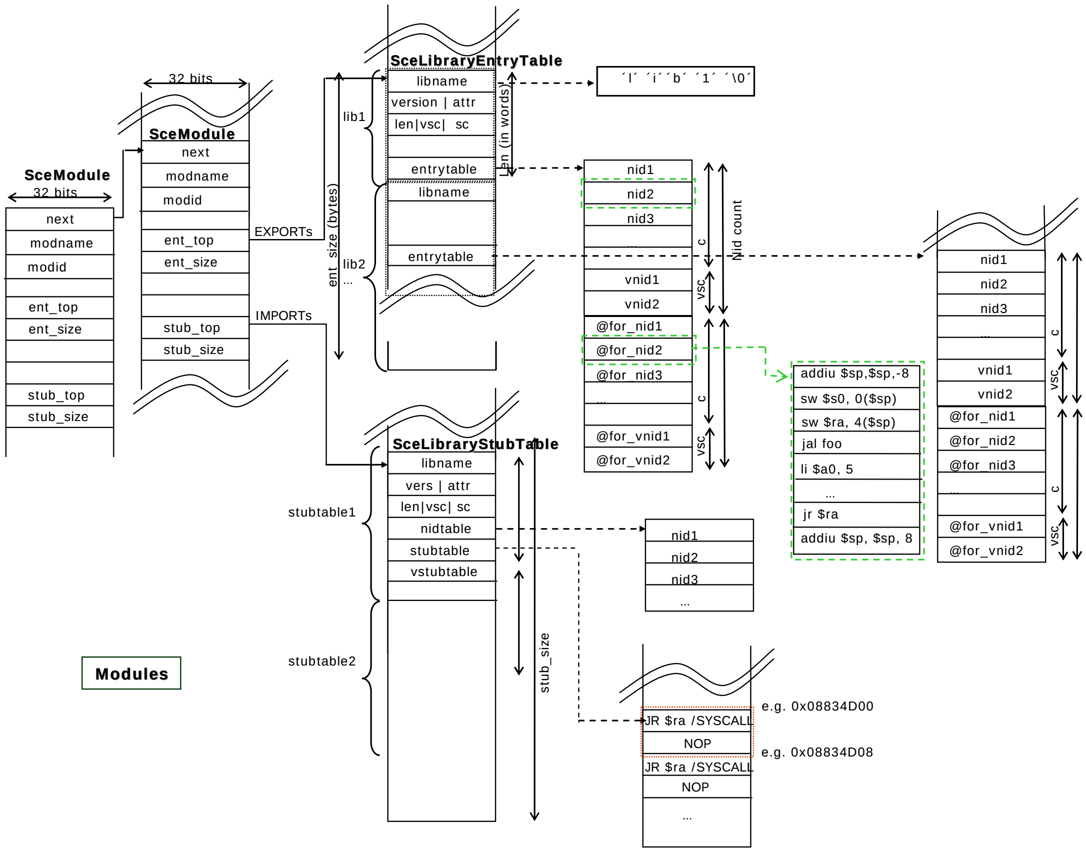
```
// you can also use
int sceKernelModuleCount(void);
```

C code 8-9 pspForeachLoadedModule.

8.3.8 Kernel internal management of loaded modules

For each loaded module, the kernel creates a SceModule struct in memory, containing information about the module such as its name, module attributes, module libraries and function exports and module imports, and adds it to a single linked list. The SceModule ent_top points to a memory region of size ent_size bytes where Library Entries are kept. Each library entry (represented as a SceLibraryEntry struct) manages a set of exported symbols for that library: the library name, the version and attributes and the number of exported functions and exported variables. The len (in 32-bit words) determines the total size of the SceLibraryEntry (other struct members may be added in the future, it is not safe to assume that the objects in memory are of fixed size, use the len to iterate each library). The entry_table points to a memory array where the mapping of NIDs and the corresponding functions is kept: an array of NIDs (for both the functions and variables) followed by the memory addresses of the exported functions and variables (e.g. the green box in Figure below shows the NID for the 2nd exported function of the first library and the address of the function code).

Likewise, the set of imported function stubs for the module is pointed at by the stub_top pointer, pointing to a memory region of stub_size bytes. This memory region contains a variable-length stub table for each imported library. Each Imported library is represented by a stub table (variable-sized struct) SceLibraryStubTable. The member nidtable points to a NID table (contiguous memory array) with the library imported NIDs. For each imported NID, a two-instruction stub is a placeholder that will be resolved when the module is linked (an empty stub just jumps at the return address register). As shown in the figure, address 0x08834D08 is the address of the stub for imported NID 2 of the first imported library.



Once the memory layout is known, a kernel mode application may iterate the list of loaded modules, find a given exported function of a module library and get the memory address of the exported function once loaded in memory. Similarly, the application may dynamically during run-time, change the address associated to a given library NID. A possible implementation is as follows:

```
/**
 * Find the address of a Module Export function
 *
 * @param mod - SceModule that exports the function
 * @param lib - Lirary name
 * @param nid - NID to look for.
 *
 * @return addr of the stub e.g. 0xDEADBEEF
 *             0xDEADBEEF: JAL or J $ra...)
 *             0xDEADBEF3: NOP
 */

static MemAddress
pspModuleFindExportedNidAddr (SceModule* mod, const char* lib, u32 nid)
{
return _pspModuleExportHelper (mod, lib, nid, (u32)NULL);
}




/**
 * Change the address of a Module Export function
 *
 * @param mod - SceModule that exports the function
 * @param lib - Lirary name
 * @param nid - NID to look for.
 * @param newProcAddr - new address bound to that nid (match prototype)
 *
 * @return old Proc Addr
 */
```

```
static MemAddress
pspModuleSetExportedNidToAddr (SceModule* mod, const char* lib, u32 nid, MemAddress
newProcAddr)
{
 return _pspModuleExportHelper (mod, lib, nid, newProcAddr);
}
```

C code 8-10 pspModuleSetExportedNidToAddr and pspModuleFindExportedNidAddr.

Both functions do basically the same task, so we can have a single function doing the work:

```
/**
 * Internal Helper function to retrieve / store the Proc Address
 * of a given NID of a Module Exported Library
 *
 * @param mod - SceModule that exports the function
 * @param lib - Lirary name
 * @param nid - NID to look for.
 * @param newProcAddr - new address to store. If NULL(0) just retrieve the existing one
 *
 * @return addr associated to the NID (old address if changed)
 */

static MemAddress
_pspModuleExportHelper (SceModule* mod, const char* lib, PspNID nid, MemAddress
newProcAddr)
{
 assert (mod);

 MemAddress ent_next = (MemAddress) mod->ent_top;
 MemAddress ent_end  = (MemAddress) mod->ent_top + (mod->ent_size >> 2);
// size is in bytes

 while (ent_next < ent_end)
 {
         SceLibraryEntryTable* ent = (SceLibraryEntryTable*)ent_next;

         if (ent->libname && strcmp(ent->libname, lib) == 0)
         {
                 int count = ent->stubcount + ent->vstubcount;
                 PspNID* nidtable = (PspNID*)ent->entrytable;
                 int i;
                 for (i = 0; i < count; i++)
                 {
                         if (nidtable[i] == nid)
                         {
                                 MemAddress procAddr =(MemAddress)nidtable[count+i];
                                 if (newProcAddr) {
                                         nidtable[count+i] = (u32)newProcAddr;
                                 }
                                 return procAddr;
                         }
                 }
                 return (MemAddress)0;
         }
         ent_next +=  ent->len;  // len in 32-bit words.
 }
 return (MemAddress)0;
}
```

Knowing this, an application in kernel mode that wishes to execute a module exported function once the module is loaded can look for the address using the above functions and call the method. This is what our client application does: executes the server function as follows:

```
typedef int (*METHOD) (int client_id);
```

```
/**
 * Performs a request to a server. Server module (server.prx) must be loaded.
 * The service is mapped to NID 0xAB55332D and takes a client id as argument
 * and returns an integer.
 *
 * @param modserver - module that exports the NID
 */

static void
AppClientDoRequest (SceModule* modserver)
{
 int client_id = 666;
 METHOD server_f = (METHOD)pspModuleFindExportedNidAddr (modserver,
         "server", 0xAB55332D);
 int result = (*server_f)(client_id);

 const char* msg;
 if (result == 1)
          msg = "good";
 else if (result == 0)
         msg = "bad";
 else
         msg = "server misbehaving";

 printf ("Client request: %s\n",msg);
}
```

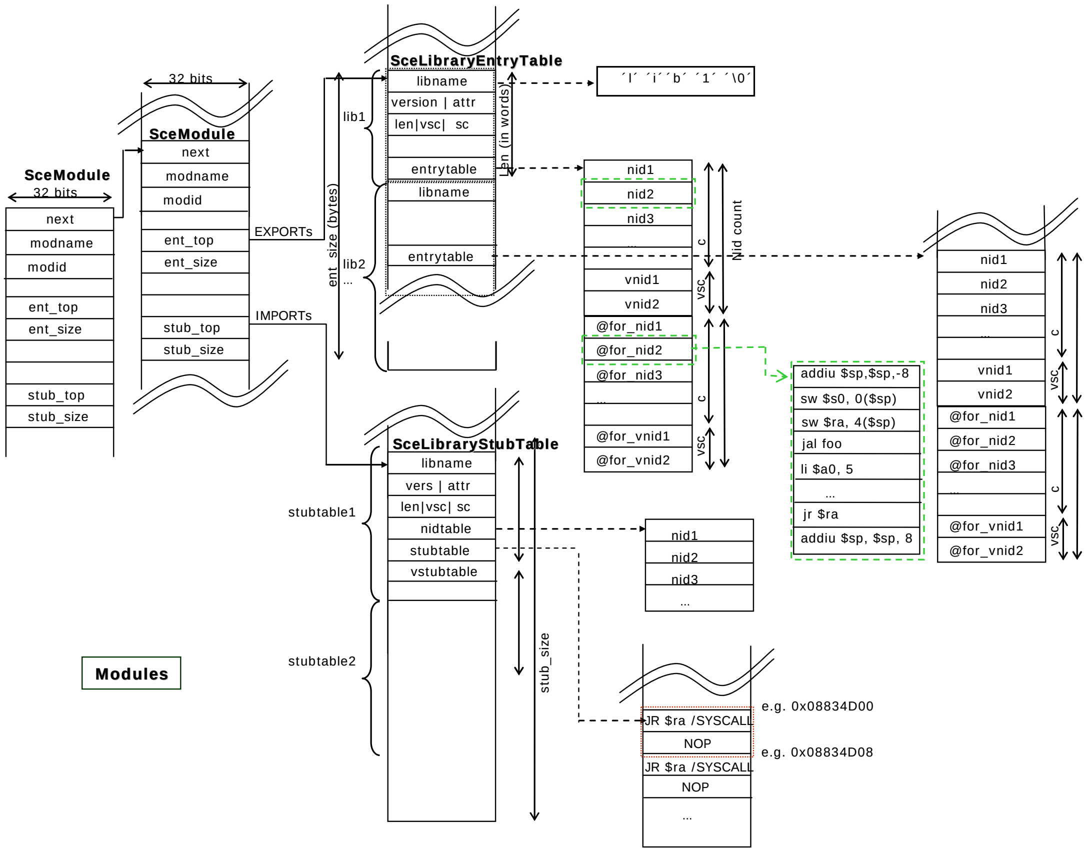
As expected, if we run this code, the client gets a result of 0 and outputs “bad”

9 “Patching” modules

By design, a client will not be granted privileges by the supervisor, unless its identifier is 123. Assume, for any reason, that the client module coder wants to access to the server service, and unfortunately:

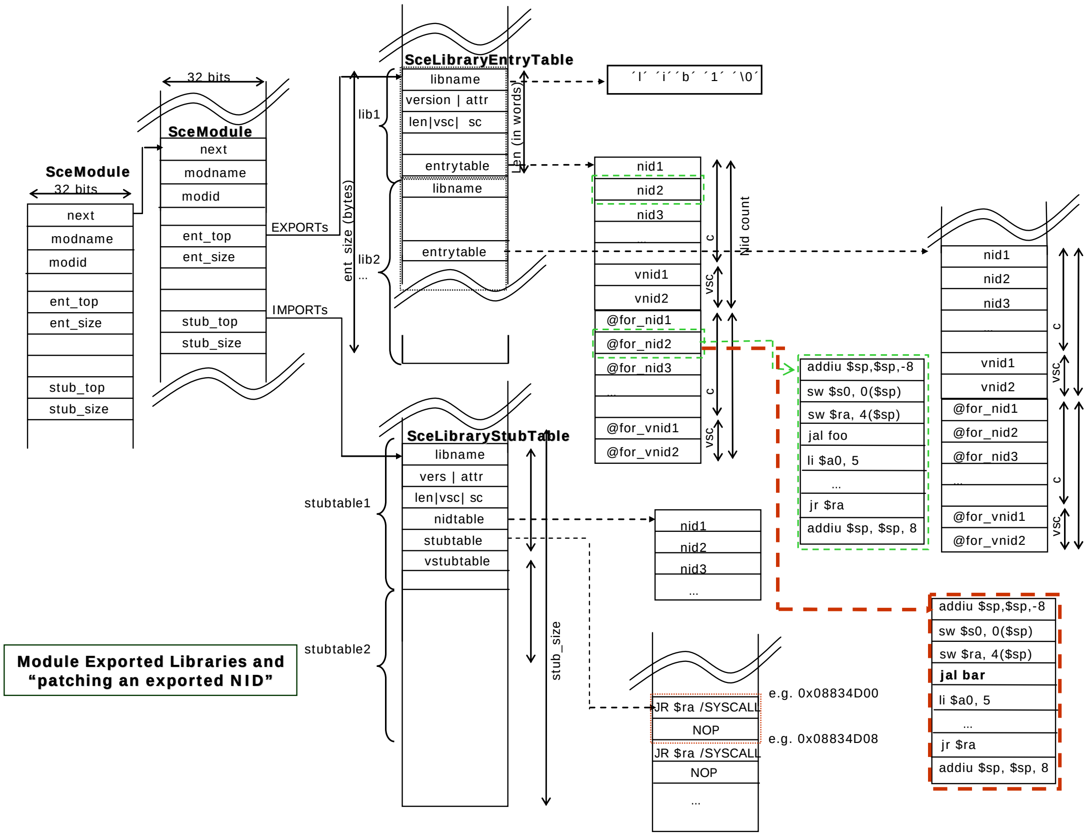
- It is not possible to change the identifier.
- We have no access to the server and supervisor modules source code.
- We cannot modify the supervisor and server modules PRX files.

A possible approach in this case (done, for example, in the SDK in a few places and by custom firmwares) is to modify the modules once in memory, so they behave as we want them to. This is not always possible (for example, an operating system may mark some memory regions “read-only”). In the following, we will look into the details on how the kernel manages the modules, and how we can modify the behavior of the system dynamically. This is commonly known as “patching”. For this, all presented methods are mainly based on accessing memory addresses and changing pointer values or modify/construct MIPS instructions.



10 Patching the Server Module Export Table

A first approach consists in patching the server export table, once it has been loaded in memory. We have just detailed how the kernel maps NIDs to function addresses. The main idea is that we can change the values in memory, so a given library NID is resolved to an address of a function that fully replaces the exported function.



Module Exported Libraries and “patching an exported NID”

Our `server_process_req` replacement is simple:

```



int
server_process_req_patched (int id)
{
 sceKernelDelayThread (10000);
 return 100;
}
```

The client just needs to do the following:

```
// Store the addresses of original procs.
METHOD    sv_chech_permissions_orig    =    (METHOD)pspModuleFindExportedNidAddr    (modsuper,
"supervisor", 0xB2C390CB);
METHOD   server_process_req_orig        =    (METHOD)pspModuleFindExportedNidAddr   (modserver,
"server",      0xAB55332D);

printf ("Orig Super Process Method: %p\n", (void*)sv_chech_permissions_orig);
printf ("Orig Server Process Method: %p\n", (void*)server_process_req_orig);
(void)sv_chech_permissions_orig;  // In case we need to use it in our patched file

// Use Server module as supposed to.
AppClientDoRequest (modserver);

// Trivially,change the whole procedure
pspModuleSetExportedNidToAddr          (modserver,          "server",            0xAB55332D,
(MemAddress)server_process_req_patched);

AppClientDoRequest (modserver);
```

Note that the same process could be applied to the supervisor module, so the supervisor exported function sv_check_permissions could be replaced completely. As a side effect, all calls from loaded modules to this exported function would call our patched version. To make things more interesting, we will add another constraint: we want the supervisor module to

```
grant us access only when called from the server module, and we want the default behavior
in other cases. Let’s see how this can be done.
```

11 Patching the Server “server_process_req” exported method.

The previous solution, although interesting, is not appropriate for our purposes, because we need to replace the whole “`server_process_req`” function and we need to duplicate the functionality that the server performs. This may not be easy.

Once we have resolved the address of the exported function given a module, a library and a NID, we can change specific instructions so they behave as we want. In the case shown in the figure above, once we know the address of the function, by inspecting the PRX dump of the server module, we can find the offset to the JAL instruction where the call to `sv_check_permissions` exported function (in the Figure we could find the address of the `jal foo` call and change it to `jal bar`. (if you look at the figure, it would be like changing the jal foo instruction inside the green box, not making the library point to the red box.

```
MemAddress      baseaddr;
MemAddress      calladdr; // absolute addressing.
MIPSInstruction newinstr;

// Using the base address of the server_process_req and the
// offset to the "jla supervisor_0xB2C390CB", compute the
// memory address to be patched / changed.
baseaddr = (MemAddress) server_process_req_orig;
calladdr = baseaddr + (0x18 >> 2);
newinstr = JAL_TO(sv_check_permissions_patched);

printf("Patch address: 0x%08X with op: 0x%08X\n", (u32)calladdr, (u32)newinstr);
// dump before
printf ("before patch\n");
pspDumpMemRegion (calladdr - 2, calladdr + 1);

// patch
int intc = pspSdkDisableInterrupts();
_sw(newinstr, (u32)calladdr);

sceKernelDcacheWritebackAll();
sceKernelIcacheInvalidateAll();
pspSdkEnableInterrupts(intc);

// dump after
printf ("after patch\n");
pspDumpMemRegion (calladdr - 2, calladdr + 1);
```

12 Patching the Server Module Import (stub) Table

Finally, patching individual instructions inside the `server_process_req` may not be appropriate: the supervisor `sv_check_permissions` method may be called several places inside or outside the `server_process_req`. In some cases it may be enough, when we want to change the behavior locally, in some cases only.

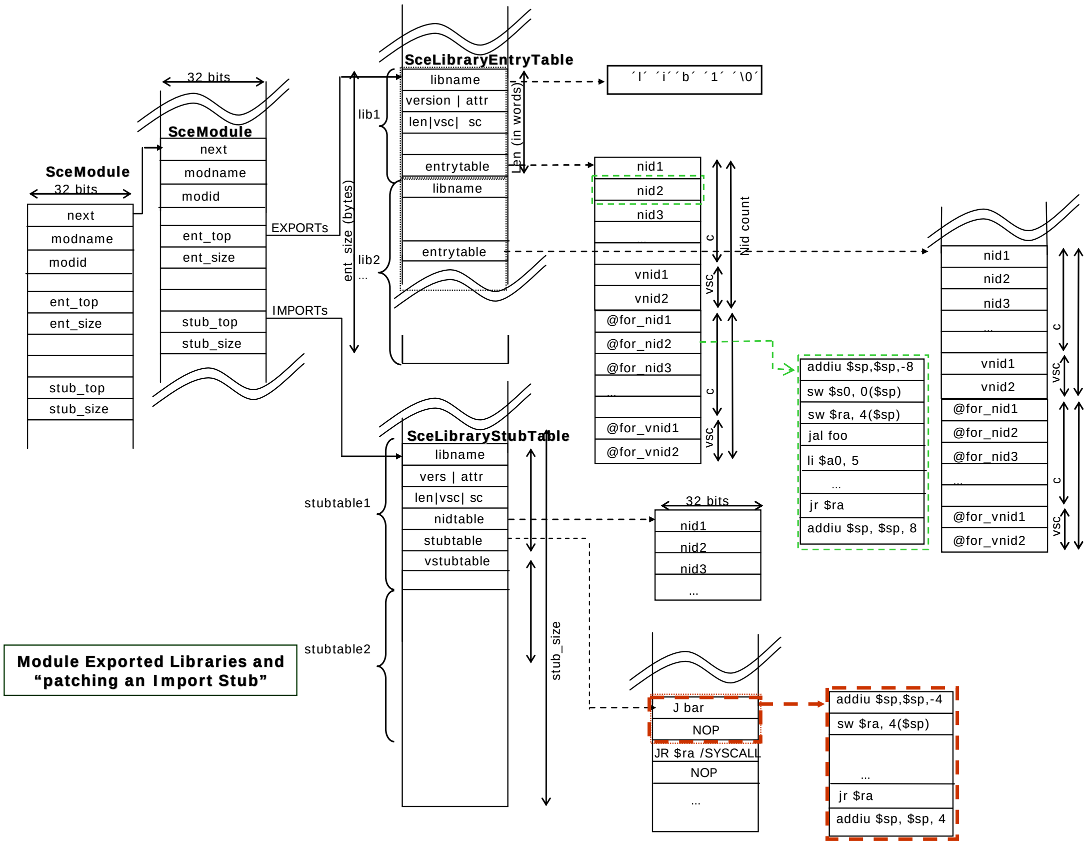
When it is not easy to find all the cases where the server module calls the supervisor exported functions, another approach is to patch the stub import table: we know that the kernel, given a loaded module, and for each imported library, the kernel manages a table with addresses to the “stubs”, placeholders where usually a direct jump to the imported function is stored (or a syscall).

Applying the same reasoning as we did before, we can find the address of the two-instruction offset of an imported function for a given module. A possible implementation is as follows:

```
/**
 * Find the address of a Module Import stub given the NID
 *
 * @param mod - SceModule that imports the NID
 * @param nid  - NID to look for.
 *
 * @return addr of the stub e.g. 0xDEADBEEF
 *              0xDEADBEEF: J $ra
 *              0xDEADBEF3: NOP
 */
static MemAddress
pspModuleFindImportStubAddrByNid (SceModule* mod, PspNID nid)
{
 assert(mod);
 MemAddress stub_next =  (MemAddress) mod->stub_top;
MemAddress stub_end  =  (MemAddress) mod->stub_top + (mod->stub_size >> 2);

while (stub_next < stub_end)
{
        SceLibraryStubTable* stub = (SceLibraryStubTable*) stub_next;
        u32* table = stub->stubtable;
        int n = stub->stubcount;
        int j;
        for (j = 0; j < n; j++)
        {
                if (stub->nidtable[j] == nid)
                {
                        dbg_log ("%-3d: UID 0x%08X, Addr 0x%08X\n",
                                j+1, stub->nidtable[j], (u32)& table[j << 1]);
                        return ((MemAddress) &(table[j << 1]));
                }
        }
        stub_next += stub->len; // len in 32-bit words.
}
return (MemAddress)0;
}
```

Graphically, the patch process is shown in the Figure below: we replace the stub entry for imported NID supervisor_B2C390CB with a direct call to our patched modified version. For this, we just construct dynamically a direct J instruction to our address and put a NOP for the delay instruction. As long as our patched method has the same arguments and return address, and considering what we discussed about the differences between J and JAL, once our method executed, we will jump to the right return address.

```
MemAddress      baseaddr;
MemAddress      calladdr; // absolute addressing.
MIPSInstruction newinstr;

MemAddress sv_stub_JR_addr = pspModuleFindImportStubAddrByNid (modserver,);
baseaddr = sv_stub_JR_addr;
calladdr = baseaddr;
newinstr = J_TO(sv_check_permissions_patched);

printf("Patch address: 0x%08X with op: 0x%08X\n", (u32)calladdr, (u32)newinstr);
// dump before
printf ("before patch\n");
pspDumpMemRegion (calladdr - 2, calladdr + 1);

// patch
int intc = pspSdkDisableInterrupts();
_sw(newinstr, (u32)calladdr);
_sw(NOP,   (u32) (calladdr + 0x01));

sceKernelDcacheWritebackAll();
sceKernelIcacheInvalidateAll();
pspSdkEnableInterrupts(intc);

// dump after
printf ("after patch\n");
pspDumpMemRegion (calladdr - 2, calladdr + 1);
```



Module Exported Libraries and “patching an Import Stub”

```
#define PATCH_PROC         0
#define PATCH_IMPORTED_NID 1

int
main(void)
{
 //const int choice = PATCH_PROC;
 const int choice = PATCH_IMPORTED_NID;

 pspDebugScreenInit();
 pspSdkInstallNoDeviceCheckPatch();
 pspSdkInstallNoPlainModuleCheckPatch();
 pspSdkInstallKernelLoadModulePatch();
 AppSetupExitCallback();

 SceModule* modsuper  = pspModuleLoadStartInKernelPart ("supervisor.prx");
 SceModule* modserver = pspModuleLoadStartInKernelPart ("server.prx");

 pspForeachLoadedModule (print_module);

 // Store the addresses of orignal procs.
 METHOD   sv_chech_permissions_orig    =   (METHOD)pspModuleFindExportedNidAddr   (modsuper,
"supervisor", 0xB2C390CB);
 METHOD  server_process_req_orig       =   (METHOD)pspModuleFindExportedNidAddr  (modserver,
"server",     0xAB55332D);

 printf ("Orig Super Process Method: %p\n", (void*)sv_chech_permissions_orig);
 printf ("Orig Server Process Method: %p\n", (void*)server_process_req_orig);
 (void)sv_chech_permissions_orig;  // In case we need to use it in our patched file

 // Use Server module as supposed to.
 AppClientDoRequest (modserver);

 // Trivially,change the whole procedure
 pspModuleSetExportedNidToAddr          (modserver,          "server",          0xAB55332D,
(MemAddress)server_process_req_patched);
 AppClientDoRequest (modserver);
// Set as it was
pspModuleSetExportedNidToAddr          (modserver,          "server",          0xAB55332D,
(MemAddress)server_process_req_orig);
AppClientDoRequest (modserver);

MemAddress      baseaddr;
MemAddress      calladdr; // absolute addressing.
MIPSInstruction newinstr;

if (choice == PATCH_PROC)
{
        // Using the base address of the server_process_req and the
        // offset to the "jla supervisor_0xB2C390CB", compute the
        // memory address to be patched / changed.
        baseaddr = (MemAddress) server_process_req_orig;
        calladdr = baseaddr + (0x18 >> 2);
        newinstr = JAL_TO(sv_check_permissions_patched);
}
else if (choice == PATCH_IMPORTED_NID)
{
        MemAddress    sv_stub_JR_addr    =    pspModuleFindImportStubAddrByNid    (modserver,
0xB2C390CB);
        baseaddr = sv_stub_JR_addr;
        calladdr = baseaddr;
        newinstr = J_TO(sv_check_permissions_patched);
}

printf("Patch address: 0x%08X with op: 0x%08X\n", (u32)calladdr, (u32)newinstr);
// dump before
printf ("before patch\n");
pspDumpMemRegion (calladdr - 2, calladdr + 1);

// patch
int intc = pspSdkDisableInterrupts();
_sw(newinstr, (u32)calladdr);
if (choice == PATCH_IMPORTED_NID) {
         // newinstr = LUI(0x00, 0x00);
         // newinstr = NOP;
        _sw(NOP,   (u32) (calladdr + 0x01));
}
sceKernelDcacheWritebackAll();
sceKernelIcacheInvalidateAll();
pspSdkEnableInterrupts(intc);

// dump after
printf ("after patch\n");
pspDumpMemRegion (calladdr - 2, calladdr + 1);

AppClientDoRequest (modserver);
printf("Done\n");
sceKernelExitDeleteThread(0);
return 0;
}
```

13 Conclusions

We have explained how PSP modules can be created, loaded, and patched. Similar approaches are being done in the SDK and in custom firmwares: for example, the kernel module responsible of loading and starting modules may check the device from which a given module is loaded, and deny the loading of modules from the memory stick (the supervisor checks the policy “not loaded from MS” and the server, representing the “load module” service, denies the request). By patching the address or function call where this is checked, we can replace the check by a “true” regardless of the device.

Similarly, it would be possible to hide applications from the XMB as it is done in custom firmwares to hide the “corrupt icons”: the directory open and directory read function addresses used to read the entries of directories can be found, and wrapper functions used, in such a way that the original function is not called in some cases, or called twice.