Sony PlayStation Vita 3.65 / 3.67 / 3.68 - 'h-encore' kernel and user modifications

Archived security papers and articles in various languages.

EDB-ID: 45377	**Author**: TheFloW (https://www.exploit-db.com/author/?a=9679)	**Published**: 11
Type: Papers (https://www.exploit-db.com/papers/)	**Platform**: Hardware (https://www.exploit-db.com/platform/?p=Hardware)	**Language**: (https://www.exploit-db.com/pap
Advisory/Source: Link (https://github.com/TheOfficialFloW/h-encore/blob/85b345e28f38d728a7ac33a35b9d6f32db93e8fc/WRITE-UP.md)	**Paper**: Download (https://www.exploit-db.com/download/45377.md) / View Raw (https://www.exploit-db.com/raw/45377/)	

« Previous Paper (https://www.exploit-db.com/docs/english/45374-xml-external-entity-injection---explanation-and-exploitation.pdf) Next Paper » (0)

```
# h-encore write-up

*h-encore*, where *h* stands for hacks and homebrews, is the second public jailbreak for the *PS Vita™* which supports the newest firmwares 3.65, 3.67 and 3.68. It allows you to make kernel- and user-modifications, change the clock speed, install plugins, run homebrews and much more.

## Introduction

Welcome to my write-up of the *h-encore* exploit chain. In this write-up I want to introduce you the security measures of the PS Vita, explain how I found the vulnerabilites, how I exploited them and how I bypassed the measures by using modern techniques.
I hope that it is an interesting read for both beginners and experts, and for both people in the scene and out of the scene.

## Userland exploit

*h-encore* uses a different entry point than its predecessor [HENkaku](https://github.com/henkaku/henkaku). Instead of a WebKit exploit, it is using a gamesave exploit. The reason for that is after firmware 3.30 or so, Sony introduced `sceKernelInhibitLoadingModule()` in their browser, which prevented us from loading additional modules. This limitation is crucial, since this was the only way we could get more syscalls (than the browser uses), as they are randomized at boot and only assigned to syscall slots if any user module imports them.

The reason why a gamesave exploit is possible on such a system is because games that were developed with an
```

SDK 2.60 and lower were compiled as a statically linked executable, thus their loading address is always the same, namely `0x81000000`, and they cannot be relocated to an other region. They also don't have stack protection enabled by default, which means that if we can stack smash in such a game, we can happily do ROP.

Finding a gamesave exploit

Initially I have implemented the kernel exploit on top of a different gamesave exploit that I have found a few years ago. This game is however expensive and takes a while in-game until it can trigger the exploit. Hence it is not suitable for the public. To reach a bigger audience, I wanted to have a [drm-free demo](https://wololo.net/talk/viewtopic.php?t=38342) as user entry point, since they can be installed on any device using [psvimgtools](https://github.com/yifanlu/psvimgtools).
Looking for gamesave exploits is a boring process, you just fuzz gamesaves by writing random stuff at random locations until you get a crash (best bet is extending strings and hope that you can smash the stack), so I asked [Freakler](https://twitter.com/freakler94) to help me on that. He then found a promising crash in the game bittersmile which was caused by a write instruction where the content and destination could be controlled.

Buffer overflow

The bug relies on the parser of the bittersmile game. The gamesave is actually a text file and the game reads it line by line and copies them to a list of buffers. However it doesn't validate the length, thus if we put the delimiter `\n` far away such that the line is longer than the buffer can hold, we get a classic buffer overflow.
If this buffer is on stack, we can make it overwrite the return address and straightly execute our ROP chain. However it is on the data section, but luckily for us, the content after the buffer is actually the list that contained destinations for other lines. This means that if we overflow into the list and redirect the buffer, we can copy the next line to wherever we want and therefore enable us an arbitrary write primitive (see [generate.py](https://github.com/TheOfficialFloW/h-encore/blob/master/scripts/generate.py)).

Partial ASLR

It seems like ASLR is only enabled for executables/modules. Normally this is not an issue, since the executable is the first thing allocated and its randomization will propagate and affect subsequent allocations (for example thread stacks).
But as our executable is statically loaded, all other allocations will be deterministic. Hence we can choose our destination address to be in a thread stack and overwrite a return address.

Return-oriented programming

The PS Vita prevents us from executing code in data sections, thus we must apply a technique called "return-oriented programming" or briefly ROP to run code. The technique is clever: since we can only execute code that is in a text section, we will simply use existing code to perform whatever we want. How does this work?
When we call subroutines, the link register `lr`, holding the return address, is pushed onto stack on enter, and popped from it on exit in order to return back to the point where it was called.
Therefore, if we manage to overwrite the return address, we can control the program counter `pc` and make it return to anywhere we want. If this "somewhere" consists of some useful instructions which end in a `pop {..., pc}`, we can again control the return address and point to the next "gadget". By induction this can be continued forever (until we run out of stack) and that's what we call a ROP chain.

Writing ROP chains

Static ROP chains can be produced using any scripting language or even manually with a hex editor, as it is only a matter of chaining gadgets together. Normally one would only write a little chain to call `system()`

or `mprotect()`. Unfortunately it is prohibited to allocate executable memory in an unprivileged process, consequently whatever we want to achieve must be implemented entirely in ROP.
I decided to use the GNU Assembler to compile my ROP chain, since it is flexible and allows us to precisely set up data. It is also possible to define macros that can be used to implement certain functionalities by smaller ROP chains (see [macros.S](https://github.com/TheOfficialFloW/h-encore/blob/master/include/macros.S)).

These macros allow us to express ROP code in a simple and readable fashion. Consider this code in C:

```c
  // Load stage2
  system_dat_fd = sceIoOpen(savedata0_system_dat_path, SCE_O_RDONLY, 0);
  sceIoRead(system_dat_fd, STAGE2_ADDRESS - (0x48 + 0x4 + _end - _start), STAGE2_SIZE);
  sceIoClose(system_dat_fd);
```

Using macros, it can be implemented as follows in ROP. This is a code snippet of stage1 which loads stage2 to a bigger stack (see [stage1.S](https://github.com/TheOfficialFloW/h-encore/blob/master/stage1/stage1.S)):

```c
  // Load stage2
  call_vvv sceIoOpen,  savedata0_system_dat_path, SCE_O_RDONLY, 0
  store_rv ret,        system_dat_fd
  call_lvv sceIoRead,  system_dat_fd, STAGE2_ADDRESS - (0x48 + 0x4 + _end - _start), STAGE2_SIZE
  call_l   sceIoClose, system_dat_fd
```

where `l` means that the argument will be loaded/dereferenced, `v` that it is a value/constant and `r` that it is the return value (in ARM the return value is written to the register `r0`).

Kernel exploit

The kernel vulnerability, that is being used in *h-encore*, is within the SceNgs module, an audio engine designed for games to make sound effects. The vulnerability has been discovered on the 2018-02-04 and was successfully exploited four days later.

Design flaws

While looking at the SceNgs module I came across the syscalls `sceNgsVoiceDefGet*()` which returned "encrypted" kernel pointers that were xor'ed with the value `0x9e28dcce`. This is a very bad design and instead of this, Sony should have returned an UID or so. After a quick search for this value, I learned that it was used by the syscalls `sceNgsRackGetRequiredMemorySize()` and `sceNgsRackInit()`. One of their parameters is a struct of type `SceNgsRackDescription` that requires the field `pVoiceDefn` be set to any of these encrypted pointers.

Below are the said structures (uninteresting fields have been omitted):

```c
typedef struct SceNgsVoiceDefinition {
  SceUInt32 uMagic;                // 0x00
  SceUInt32 uFlags;                // 0x04
  SceUInt32 uSize;                 // 0x08
  SceInt32  nSomeOffset;           // 0x0c
  // ...
```

```
  SceInt32  nVoicePresetsOffset;    // 0x30
  SceUInt32 uNumVoicePresets;       // 0x34
  // ...
} SceNgsVoiceDefinition;            // 0x40

typedef struct SceNgsRackDescription {
  const struct SceNgsVoiceDefinition *pVoiceDefn;
  // ...
} SceNgsRackDescription;
```

And here is a pseudo-code snippet of how these structures are read:

```c
  SceNgsRackDescription rackDesc;
  SceNgsVoiceDefinition voiceDefn;
  // ...
  if (!copyin(&rackDesc, pRackDesc, sizeof(SceNgsRackDescription), 1)) {
    goto error_invalid_param;
  }
  if (rackDesc.pVoiceDefn == NULL) {
    goto error_invalid_param;
  }
  rackDesc.pVoiceDefn ^= SCE_NGS_VOICE_DEFINITION_XOR;
  if (rackDesc.pVoiceDefn == NULL) {
    goto error_invalid_param;
  }
  if (!is_valid_vaddr(rackDesc.pVoiceDefn, sizeof(SceNgsVoiceDefinition))) {
    goto error_invalid_param;
  }
  if (!copyin(&voiceDefn, rackDesc.pVoiceDefn, sizeof(SceNgsVoiceDefinition), 0)) {
    goto error_invalid_param;
  }
  if (voiceDefn.uMagic != SCE_NGS_VOICE_DEFINITION_MAGIC) {
    goto error_invalid_param;
  }
  // ...
  error_invalid_param:
    return SCE_NGS_ERROR_INVALID_PARAM;
```

As we can see, `pRackDesc` is copied from user to kernel, then the field `pVoiceDefn` is decrypted. It is even checking if the kernel pointer is valid, which is to our advantage as we will see later. Finally, if all fields are valid, it will continue working with `voiceDefn`.
Since we control `pRackDesc->pVoiceDefn` and we know the xor value, it is possible to pass a fake voice definition, provided it is in kernel memory. The reason for that is the PS Vita has got SMEP/SMAP equivalent mitigation measures that prevent the kernel from implicitly reading/writing user memory or executing user executable code.

Insufficient checkings

The rack syscall is really big and almost all fields in the voice definition are uninteresting and cannot be exploited, except `nVoicePresetsOffset` and `uNumVoicePresets`.
They hold information to a list of presets of the following type (note that all offsets are relative to the preset and the offset to the preset is relative to the voice definition):

```c
typedef struct SceNgsVoicePreset {
  SceInt32  nNameOffset;
  SceUInt32 uNameLength;
  SceInt32  nPresetDataOffset;
  SceUInt32 uSizePresetData;
  SceInt32  nBypassFlagsOffset;
  SceUInt32 uNumBypassFlags;
} SceNgsVoicePreset;
```

Below is an interesting pseudo-code snippet of the gigantic syscall:

```c
  void *SceNgsBlock;
  SceNgsVoicePreset *presets;
  // ...
  // SceNgsBlock is allocated with a size determined by all presets and some other fields
  // ...
  SceNgsBlock += 0x148;
  SceNgsBlock += (voiceDefn.nNumVoicePresets * sizeof(SceNgsVoicePreset));
  presets = (SceNgsVoicePreset *)((char *)voiceDefn + voiceDefn->nVoicePresetsOffset);

  for (size_t i = 0; i < voiceDefn->uNumVoicePresets; i++) {
    // ...
    if (presets[i].nPresetDataOffset != 0 &&
        presets[i].uSizePresetData != 0) {
      memcpy(SceNgsBlock, &presets[i] + presets[i].nPresetDataOffset, presets[i].uSizePresetData);
      SceNgsBlock += presets[i].uSizePresetData;
    }
    // ...
  }
```

It is important to mention that no check on `uSizePresetData` is being made when summing up the preset sizes. Therefore a possible attack is to use a negative size on the last preset such that the resulting allocation size would be smaller than intended. This is a heap overflow primitive that we gain. For example having presets with size 0x1000, 0x1000 and -0x1000 will result in an allocation of size 0x1000 only, and while copying, the second preset should overflow the buffer.
The problem here is obviously how to stop `memcpy()` on the negative size, since a negative size as unsigned integer is actually very large (-1 == 0xFFFFFFFF).
I was inspired by a similar technique explained by [CTurt](https://cturt.github.io/dlclose-overflow.html) who proposed using a different thread to stop the overflow. However before I could try this, I had already made a proof of concept to confirm the heap overflow and to my surprise, it did overflow, but the third preset was not fully copied and therefore did not cause a segmentation fault.

memcpy, or more like memecpy

Yes, Sony failed to implement a correct memcpy. Instead of copying a huge load of data when getting a negative size, it would simply copy a few bytes and terminate. Seriously I would expect bugs anywhere but in the most basic libc function. Their wrong implementation is available in both userland and kernel, maybe even in TrustZone.

Here is the pseudo-code of their implementation. I assume that it was written in assembly rather than C

code, since it contains highly optimized instructions.

```c
void memcpy(void *dst, const void *src, size_t len) {
  void *end;

  if (len >= 32) {
    end = dst + len;                          // [1]
    while ((uintptr_t)dst & 3) {
      *(uint8_t *)dst = *(uint8_t *)src;
      src += sizeof(uint8_t);
      dst += sizeof(uint8_t);
      len -= sizeof(uint8_t);
    }
    while ((uintptr_t)dst & 31) {
      *(uint32_t *)dst = *(uint32_t *)src;
      src += sizeof(uint32_t);
      dst += sizeof(uint32_t);
      len -= sizeof(uint32_t);
    }
    if (end >= dst + 32) {                    // [2]
      // vectorized copy
      // ...
    }
  }
  while ((int32_t)len >= sizeof(uint32_t)) { // [3]
    *(uint32_t *)dst = *(uint32_t *)src;
    src += sizeof(uint32_t);
    dst += sizeof(uint32_t);
    len -= sizeof(uint32_t);
  }
  if (len & 2) {
    *(uint16_t *)dst = *(uint16_t *)src;
    src += sizeof(uint16_t);
    dst += sizeof(uint16_t);
  }
  if (len & 1) {
    *(uint8_t *)dst = *(uint8_t *)src;
  }
}
```

- The first bug is at [1]. If `len` is negative, the addition with `dst` will yield a value smaller than `dst ` due to an integer overflow and as a consequence, the comparison at [2] will result in false, no matter if it is a signed or unsigned comparison, and thus it believes that there are less than 32 bytes to copy.
- The second bug is at [3] where the length is compared as signed integer. Hence a negative length will simply bypass the copy loop.

Thanks to these bugs we can exploit the previous vulnerability differently by using a negative length to iterate backwards and therefore enable an out-of-bounds write primitive.

Exploitation

Bypassing SMAP

Our goal is to control the content of `pVoiceDefn`, however since the kernel cannot implicitly read user memory, we need to plant our data in kernel memory. This is an easy task: because of SMAP, syscalls use `copyin()` to copy user buffers to kernel stack to later access them. On returning to user the kernel stack is not cleared due to performance reasons, thus whatever data that was copied from user will remain there. A potential syscall that we can use for that purpose is `sceIoDevctl()` which can copy upto 0x3ff bytes to kernel stack (what this syscall is actually doing is not relevant and we can just pass some invalid inputs to make it return error). This is big enough to hold our voice definition and more importantly it is deeply located inside the kernel stack such that subsequent syscalls cannot corrupt it. But where is the kernel stack?

Getting the kernel stack base address

The `is_valid_vaddr()` check is useful because in case we pass an invalid kernel pointer, it will not cause a segmentation fault but return an error. Thanks to this we can set up our fake voice definition and plant it in kernel stack, then find its address by iterating through the kernel memory using `sceNgsRackGetRequiredMemorySize()`:

```c
int ret = 0;
uint32_t kstack_base = KSTACK_BASE_START;

do {
  rack_desc.pVoiceDefn = (struct SceNgsVoiceDefinition *)(kstack_base + DEVCTL_STACK_FRAME) ^
SCE_NGS_VOICE_DEFINITION_XOR;
  ret = sceNgsRackGetRequiredMemorySize(sys_handle, &rack_desc, &buffer_info.size);
  if (ret == SCE_NGS_ERROR_INVALID_PARAM)
    kstack_base += KSTACK_BASE_STEP;
} while (ret == SCE_NGS_ERROR_INVALID_PARAM);
```

Implementing a conditional loop in ROP

Writing ROP is is not as simple as writing assembly. One instruction may only be implemented with more than a dozen of gadgets. Therefore we would like to keep our code as simple as possible and eliminate the redunant addition and condition within the loop:

```c
int ret = 0;
uint32_t kstack_base = KSTACK_BASE_START - KSTACK_BASE_STEP + KSTACK_DEVCTL_INDATA_OFFSET;

do {
  kstack_base += KSTACK_BASE_STEP;
  rack_desc.pVoiceDefn = (struct SceNgsVoiceDefinition *)(kstack_base ^ SCE_NGS_VOICE_DEFINITION_XOR);
  ret = sceNgsRackGetRequiredMemorySize(sys_handle, &rack_desc, &buffer_info.size);
} while (ret == SCE_NGS_ERROR_INVALID_PARAM);

kstack_base -= KSTACK_DEVCTL_INDATA_OFFSET;
```

Now if we want to implement this in ROP we can of course not use branch instructions, as they set the program counter, not the stack pointer. What we need is a way to conditional move the stack pointer like this:

```c
```

```
if (ret == SCE_NGS_ERROR_INVALID_PARAM)
  sp = loop_start;
else
  sp = loop_end;
```

In ARM there exists the `it (if-then)` instruction which allows us to conditional move a register like `it eq ; moveq r0, r1`. Unfortunately there is no such gadget available in our executable.
However, we have got this gadget here:

```c
int cmp_eq(int r0, int r1) {
  if (r0 == r1)
    return 1;
  else
    return 0;
}
```

Let's see how we can use it to realize our instruction by exploiting the zero property and identity property of multiplication. Previously we wanted to set `sp` to either `loop_start` or `loop_end` - this time we use a temporary register and set it to either "something" or "nothing":

```c
if (ret == SCE_NGS_ERROR_INVALID_PARAM)
  tmp = loop_start - loop_end;
else
  tmp = 0;
sp = tmp + loop_end;
```

The mentioned properties say that the product of any number times zero is zero and the product of any number times one is itself, therefore if we can set `tmp ` as the product of `cmp_eq(ret, SCE_NGS_ERROR_INVALID_PARAM)` times `loop_start - loop_end`, we get our desired logic:

```c
sp = (cmp_eq(ret, SCE_NGS_ERROR_INVALID_PARAM) * (loop_start - loop_end)) + loop_end;
```

Note that this is only one of the few possibilities and we could also have realized it by exploiting the properties of logical conjunction:

```c
sp = ((cmp_eq(ret, SCE_NGS_ERROR_INVALID_PARAM) - 1) & (loop_end - loop_start)) + loop_start;
```

Finally in ROP it looks like this (it's cute but check out [mul_add_rvv](https://github.com/TheOfficialFloW/h-encore/blob/master/include/macros.S#L143)):

```c
  // Iterate through kernel memory
  loop_start:
    // kstack_base += KSTACK_BASE_STEP
    load_add_store kstack_base, kstack_base, KSTACK_BASE_STEP
```

```
    // rack_desc.pVoiceDefn = kstack_base ^ SCE_NGS_VOICE_DEFINITION_XOR
    xor_rv   ret, SCE_NGS_VOICE_DEFINITION_XOR
    store_rv ret, rack_desc + 0x00

    // Call sceNgsRackGetRequiredMemorySize on the xor'ed kstack_base
    load_call_lvv_2 sceNgsRackGetRequiredMemorySize, sys_handle, rack_desc, buffer_info + 0x04

    // Compare ret with SCE_NGS_ERROR_INVALID_PARAM and return 1 if equal, else 0
    cmp_eq_rv ret, SCE_NGS_ERROR_INVALID_PARAM

    // ret = ret * (loop_start - loop_end) + loop_end
    mul_add_rvv ret, loop_start - loop_end, loop_end

    // Stack pivot
    store_rv         ret, ldm_data_r0 + 0x0c
    set_r0_r2_ip_sp_lr_pc ldm_data_r0
  loop_end:
```

Note that we are using the macro `load_call_lvv_2` here instead of `load_call_lvv`. The difference is that the second version skips some words before the call:

```c
  .word add_sp_14_pop_pc                  // pc
  .word 0xDEADBEEF                        // dummy
  .word 0xDEADBEEF                        // dummy
  .word 0xDEADBEEF                        // dummy
  .word 0xDEADBEEF                        // dummy
  .word 0xDEADBEEF                        // dummy
```

This is neccessary because when calling subroutines in ROP, the stack gets corrupted due to the subroutine's stack allocations such as for the prologue, where callee saved registers are pushed onto stack (sidemark: if you call a syscall, then the user stack remains untouched). Normally this does not matter, but in a loop we want to reuse the gadgets. In our case the stack is corrupted by `push.w {r4, r5, r6, r7, r8, lr}` in `sceNgsRackGetRequiredMemorySize()`, since it is to be exact only a wrapper that calls the actual syscall `sceNgsRackGetRequiredMemorySizeInternal()`.

Defeating kernel ASLR

We have now only partially defeated kernel ASLR by learning the kernel stack base address. However to build a kernel ROP chain which we will later use, we must know at least a base address of a kernel module. This is because we cannot implicitly execute user executable memory in kernel, thus our ROP chain cannot consist of user gadgets.
While reverse engineering the codepath of the rack syscall, I found something bizarre: It is using some fields in the SceNgsBlock header in order to store temporary variables. Some of those are the parameters for a `copyout()`!
This means that we can easiely enable an arbitrary kernel read primitive by using our out-of-bounds exploit to overwrite these parameters (see [stage2.S](https://github.com/TheOfficialFloW/h-encore-private/blob/master/stage2/stage2.S)):

```c
  // Set presets information in voice definition
  store_vv PRESET_LIST_OFFSET, voice_def_buf + 0x30
  store_vv 2,                  voice_def_buf + 0x34
```

```
  // Set presets
  set_preset 0, 0,                    -(0x148 + 2 * 0x18) + COPYOUT_PARAMS_OFFSET
  set_preset 1, FAKE_COPYOUT_OFFSET, FAKE_COPYOUT_SIZE

  // Overwrite copyout's dst, src and len
  call_vvv memset,      voice_def_buf + FAKE_COPYOUT_OFFSET, 0, FAKE_COPYOUT_SIZE
  store_vv sysmem_base, voice_def_buf + FAKE_COPYOUT_OFFSET + 0x04 // dst
  add_lv   kstack_base, KSTACK_SYSMEM_OFFSET
  store_rv ret,         voice_def_buf + FAKE_COPYOUT_OFFSET + 0x08 // src
  store_vv 4,           voice_def_buf + FAKE_COPYOUT_OFFSET + 0x1c // len

  // Trigger exploit
  trigger_exploit

  // Get SceSysmem base address
  load_add_store sysmem_base, sysmem_base, SCE_SYSMEM_BASE
```

Note how the first preset is used to move the destination pointer backwards and how the second is used to overwrite the target with controlled data.

Choosing an OOB target

My first idea is to overwrite a function pointer of any object whose invocation we can control by some syscall. Our SceNgsBlock is allocated with `ksceKernelAllocMemBlock()`, hence our target must also be allocated the same way such that their memory blocks may be consecutive. I did not find anything at that time, so I asked [xyz](https://twitter.com/pomfpomfpomf3) for some advice. He reminded me that user threads possess both a user and kernel stack, and since the kernel stack is allocated with `ksceKernelAllocMemBlock()`, aiming at it is possible.

Controlling the OOB write

I spawned alot of threads and read their kernel stack base addresses to see if they were close to the SceNgsBlock and noticed a few things:

- Kernel stacks are 0x1000 bytes big and are located at odd addresses like `0xXXXX1000, 0xXXXX3000, 0xXXXX5000, etc.`.
- When the range `0xXXXX4000-0xXXXX6000 `for example is free, kernel stacks choose the upper part at `0xXXXX5000`.
- The most recently used block is taken if we free a block and allocate an other one with the same size.
- Blocks will eventually be consecutive.

Remember that our out-of-bounds write can manipulate anything that has a lower address than our block, therefore we want to bring the kernel stack behind the SceNgsBlock. A strategy to achieve this is as follows:

1. Spray alot of blocks with a size of 0x2000.
2. After many allocations the blocks will eventually become consecutive (and even).
3. Free the second last block to create a hole.
4. Spawn a thread to fill this hole with the kernel stack.
5. Free the last block and allocate our evil rack.

The spawned thread (let's call it thread 2 and the current thread 1) must now simply call any syscall. This syscall should not terminate before its stack has been overwritten. To accomplish that we can for

example acquire a semaphore in thread 2, overwrite the kernel stack in thread 1, and finally release the semaphore in thread 1 to make the acquire syscall terminate and trigger kernel ROP execution on midway. This is however more effort than actually needed and can be done in a simpler fashion by using the syscall `sceKernelDelayThread()` to make the thread wait a a little bit. I choosed 100ms which is enough time for thread 1 to hijack the kernel stack of thread 2 before it returns back to user.
The reason why it works is because the delay syscall calls some subroutines which make the stack grow (callee saved registers and the return address are pushed onto stack), then after 100ms the stack will shrink back and data will be popped from the stack. If we manage to overwrite the return address between these two phases, we can execute our kernel ROP chain. Note that since the stack grows from high to low and we are attacking from above, we don't need to deal with stack cookies.

Kernel ROP execution

As the return address that we have overwritten is very near the stack bottom (because the syscall `sceKernelDelayThread()` doesn't require alot of stack), we don't have much room for our ROP chain. Therefore we have to plant our actual kernel ROP chain somewhere else, and stack pivot to it. Just as how we have planted the fake voice definition into kernel stack, we can use `sceIoDevctl()` to plant our kernel ROP chain. As we are doing this in the same thread, we don't need to know its address and can simply move the stack pointer backwards from `kstack_base + 0x1000 - 0x64` to `kstack_base + 0x6f8` as follows (see [krop.S](https://github.com/TheOfficialFloW/h-encore/blob/master/stage2/krop.S)):

```c
  .set difference, (KSTACK_SIZE - OVERWRITE_SIZE) - KSTACK_DEVCTL_INDATA_OFFSET
  push 1, SceSysmem_pop_r3_r4_r5_r6_r7_pc             // pc
  push 1, SceSysmem_pop_r3_pc                         // r3
  push 1, SceSysmem_sub_r1_r1_r3_bx_lr                // r4
  push 9, 0xDEADBEEF                                  // r5
  push 1, SceSysmem_pop_pc                            // r6
  push 9, 0xDEADBEEF                                  // r7
  push 1, SceSysmem_add_r1_sp_bc_mov_r2_r6_blx_r3     // pc
  push 0, 0x1c + 0xbc + difference                    // r3
  push 1, SceSysmem_blx_r4_pop_r4_pc                  // pc
  push 9, 0xDEADBEEF                                  // r4
  push 1, SceSysmem_mov_sp_r1_blx_r2                  // pc
```

Note that `push` is a macro that takes a base as first argument which says what type the next argument has, where `0: constant`, `1: SceSysmem gadget`, `2: kernel stack (for RW operations)` and `9: dummy (won't be written)`.

The actual kernel ROP chain then allocates two RW memory blocks, copies the compressed kernel payload to first block and stores the decompressed payload in the second block. Finally, it marks the second block as executable, flushes caches and executes it.

Post-exploitation

The task of the kernel payload is to fix the ngs module, since it would crash upon exiting the process. This is because we were not supposed to use negative lengths and the rack syscall would behave differently. I did not figure out what was happening, as the function was way to big to fully understand it.
After stabilizing the kernel, it installs some custom syscalls and signature patches such that it can launch the bootstrap menu. This will then install and load [taiHEN](https://github.com/yifanlu/taiHEN) to apply patches to the system and allow us to run homebrews, plugins and apply other mods and tweaks.

Sony's patch

A firmware update has been released on 2018-09-11, 72 days after release of *h-encore*, which patched the kernel exploit by using the constants `0x9e28dcc0-0x9e28dce0` for the voice definitions. An arbitrary kernel pointer can therefore not be passed anymore. However the memcpy bug and the user exploit are still unpatched. I decided not to leave the memcpy part out of the write-up, as I don't think there is any other bug that relies on it.

Conclusion

Sony actually did a great job on this console, if compared to their other products, and it was only bad luck that these bugs were coincidentally at the wrong places. Finding and analyzing them was interesting and a good learning experience for me.
Although the vulnerabilities were easy to exploit, bypassing their mitigations was challenging, especially because I had not written any ROP chains before and never that huge.
I have also enjoyed working on this write-up and I hope that I can one day finish my second exploit chain and present that one to you.
Meanwhile I will be looking at the PS4 kernel :)

Credits

Thanks to [SpecterDev](https://twitter.com/SpecterDev) and [abertschi](https://twitter.com/andrinbertschi) for reading over the write-up and giving me feedback.

« Previous Paper Next Paper »