ROP Tutorial

Stack

The stack is used for static memory allocation. Read Wikipedia:
https://en.wikipedia.org/wiki/Stack-based_memory_allocation

Short explanation:

- When a function need an amount of memory, it decreases the stack pointer (SP) by that many bytes, and own the memory pointed to by SP.
- When a function no longer need that memory, it increases the stack pointer.

Function call

For the nX/U8 core, when a function calls another functions, the address of the caller is stored in the `LCSR:LR` register, and the arguments are (often) stored into the registers `R0-R15` (and occasionally pushed on the stack, although I can't recall any example right now).

For example, if the function `f` taking 1 1-byte argument in `r2`, has the structure

```
0:1234    st    r2, 08124h
0:1236    rt
```

and a snippet of code

```
0:2466    mov    r2, #3
0:2468    bl     0:1234h
0:246c    mov    r0, #0
```

is executed, the following will happen:

- At `0:2466`, `r2` is set to decimal value `3`.
- At `0:2468`, when the command `bl 0:1234h` is called,
  - The `LCSR:LR` register value is set to `0:246ch` (the address of the command right after the `bl` command) (that is, `LCSR = 0` and `LR = 246ch`)
  - The `CSR:PC` register value is set to `0:1234h`.
- Next, `0:1234` is executed.
- After 1 command, at `0:1236`, when the `rt` command is called, the value of `CSR:PC` is set to the value of `LCSR:LR` (which is `0:246ch`).
- Then, the command `0:246ch` is executed.

Recursive function

Next. When a function `a` that is called by another function `parent` needs to call yet another function `child`, it needs to remember the value of `LCSR:LR` when it was called (by `parent`). That is done by the command

```
push lr
```

which allocates 4 bytes of memory on the stack, and store the value of `lcsr` and `lr` there.

When the function returns, instead of

```
pop lr
ret
```

(which is supposed to return the value of `lr` and `lcsr` and then do the normal return), it executes

```
pop pc
```

.

For more detailed instruction, read the nX/U8 core instruction manual. But basically, the LR (2 bytes) is on the top, then the LCSR (1 byte), then an unused byte.

Note that, although the CPU allows up to 16 code/data segments, the calculators only uses 2 segments, so only the least significant bit of `lcsr` is important, all other bits are always 0. Moreover because of word alignment of `PC`, the least significant bit of the PC is always zero.

So, how does that help with arbitrary code execution?

As we observed, there are a lot of `pop pc` commands in the code. And those commands set the program counter to whatever on the top of the stack. Therefore, if we executes a `pop pc` command and can control what is in the stack, we can make the PC to jump to arbitrary location.

The remaining problem is to write a program in ROP.

Example

For an example, let's try understanding the hackstring that was used to understand how **F030** worked, taken from #286.

Note that you should lookup the command in the disassembly listing of the 570es+/991es+ ROM to understand what I'm saying.

```
<from #52 character> cv24 M 1 - 0 cv26 X - Int cs23 0 - cv24 M 1 - 0 cv26 cs4
- A 4 0 - ! cs32 0 -
```

First, convert it to hexadecimal for ease of understanding. You know where to look for a character table, for example you can use the Javascript code in #301.

```
?? ?? ?? ... (there are 52 bytes) ... ?? ?? ??
ee 54 31 ?? 30 f0 58 ?? 6a 27 30 ??
ee 54 31 ?? 30 f0 04 ?? 41 34 30 ?? 57 b6 30 ??
```

The values of `??` are not important, the behavior of the hackstring is the same for all (non-null) values of `??`.

And what does this hackstring do?

First, the 100-byte hackstring is repeated in the memory, from the input area (`8154h`) to the end of the writable memory (`8e00h`).

(for more information why does that happen, basically a `strcpy(0x81b8, 0x8154)` get called, and all the 100 bytes inside the `0x8154..0x81b8` ranges (includes `0x8154`, excludes `0x81b8`) are not null; and the `strcpy` in the calculator is implemented like this: (pseudo-code)

```
void strcpy(char* dest, char* src) {
         while (*src != NULL) {
            *dest = *src;
            ++dest;
            ++src;
      }
 }
```

Fortunately there are some non-writable (and so they're always null) memory in the range `0x8e00..0xefff`, so that doesn't loop forever.

)

That is, the byte at `8154h` has the value of the first byte in the hackstring, the byte at `8155h` has the value of the second byte in the string, ..., the byte at address `x` (`8154h <= x < 8e00h`) has the value of the `(x - 8154h) mod 100` (0-indexing) byte in the hackstring.

Then, (eventually) when the `PC` is at address `0:2768h`, `LR = 0:2768h` and `SP = 8da4h`. As you can calculate, the 4 bytes at address `8da4h .. 8da7h` has the values of the `52 .. 55th` bytes of the hackstring. (0-indexing) Now the 4 bytes `ee 54 31 ??` are on the top of the stack.

When the `pop pc` command is executed, `pc = 54eeh` and `csr = 1`. (remember the endianness)

When the command at `1:54eeh` is executed, `er0 = 0f030h` and `r2 = 58h`.

... hopefully you can deduce what will happen next, at least until the command `0:154f0h` is executed the second time. You should be able to figure out that the top of the stack at that time is

```
41 34 30 ?? 57 b6 30 ??
```

.

To understand the remaining 8 bytes of the hackstring, you need to know about some function addresses.

First, remember that a function which calls another function must start with `push lr` (there may be some commands that does not change `lr` before that) and ends with `pop pc`. So, if we make the `pc` be right after the `push lr` command, eventually a `pop pc` would be executed with the same `sp`, and the 4 bytes on the top of the stack is not changed (assume the function really want to returns to its caller). That way we can continue to keep control of the `pc`.

So, some useful functions in the calculator:

- `0:343eh`: A function, given an address pointed to with `er0`, and a number `r2`, print `r2` lines on the screen using the string pointed to by `er0`.
- `0:b654h`: A function taking no parameter, returning no parameter (i.e., `void f(void)`, blinks the cursor and waits for the user to press the key `shift` before returning.

So 8 bytes

```
41 34 30 ?? 57 b6 30 ??
```

(in the above hackstring) calls those two functions in order.

Note that I just mentioned the multiline print function is at `0:343eh`. Why using the bytes `41 34 30`? (so that `pop pc` set the `pc` to `0:3431`?

First, the actual command executed is at `0:3440`, because of word alignment, the value of `pc` is always even. (not so for `sp`, be careful!)

Now assume we set the `pc` to `0:343e`. It will push the value of `lr` on the top of the stack, and then execute some commands until `0:3478`, and then set the value of `pc` to the value of `lr` we pushed earlier. Which is not what we want. (as we can't set the `lr` to a desired value) That's also the reason why we don't like gadgets ending with `ret`.

Instead, we jump to the command **right after** the `push lr`. That way the `pop pc` will pop the top of the stack at that time, and we can control the value of `pc`.

(some notes regarding this:

(Summary: jumps to the command right after `push lr` is always correct, but other options may also be correct)

1. If we jump to a command before the `push lr`, the `pc` will often jump to some unexpected places, as the command right before `push lr` is often `pop pc` or `ret`.
2. If we jump to the `push lr` command, the value of `pc` after the corresponding `pop pc` will depends on the value of `lr` at the start of the function. Only useful if we can control the value of `lr`. A similar situation happen if we jump to the beginning of (or inside) a function returning with `ret` - we need to control the value of `lr`.
3. If we jump to the command right after the `push lr` the function is executed and the corresponding `pop pc` will pop the top of the stack. Good.
4. If we jump to some position after the `push lr` the function body is executed except some commands at the first.

Typically, option (3) is used because it's the simplest, but occasionally option (2) or (4) should be used instead if typing them into the calculator takes significantly less keystrokes. I will (try to) explain this part later if some hackstring uses that.

)

Loops

Next, about loops.

The only way to loop in return-oriented programming is to modify the value of the stack pointer `sp`. If you search for `sp` in the disassembly, you can see that the only commands modifying `sp` and is sufficiently near a (later) `rt` or `pop pc` (such that we can easily reason about what will happen if those commands (between the command modifying `sp` and the return command) is executed) are:

- `mov sp, er14` (often followed by `pop er14` and `pop pc`)
- `add sp, #...` (positive amount means pop that many bytes from the stack)

Only the first one is (currently) used for looping.

So, to loop we should:

- When `sp = A`, and a `pop pc` command is executed, execute something that doesn't modify the stack.
- Execute a gadget with `pop er14; pop pc` and put `A-2` on the top of the stack.
- Execute a gadget with `mov sp, er14; pop er14; pop pc`.

(it's possible to loop with something that does modify the stack, more information later)

When the last gadget `mov sp, er14; pop er14; pop pc` is executed, the following happens:

- `mov sp, er14`: `sp` is set to the value `A - 2`.
- `pop er14`: `sp` is increased by 2 bytes. `er14` contains whatever in that 2 bytes.
- `pop pc`: `pc` has the value at `A`.

This is an infinite loop. I have never written a conditional statement/loop, but it should be possible with some table lookup/etc.

Example:

Using the hackstring in #290:

```
<52 characters> cv24 M 1 - Fvar cv26 cv40 - Int cs23 0 - tan-1 D 0 - cs26
cv26 cs16 D 1 - cv12 = 0 - sin 2 0 - 0 cv34 Int cs23 0 - (-) cs32 0 - frac
Ans ^ cs32 0 - <2 remaining characters>
```

Hexadecimal:

```
?? ... (52 bytes) ... ??
ee 54 31 ?? 46 f0 fe ?? 6a 27 30 ?? b2 44 30 ?? 40 f0
1d 44 31 ?? e2 3d 30 ?? a0 32 30 ?? 30 f8
6a 27 30 ?? 60 b6 30 ?? ae 8b 5e b6 30 ??
?? ??
```

Only the 10 last bytes are related to looping.

`60 b6 30 ??` corresponds to the address `0:b660h`. The commands from `0:b660h` to `0:b662h` are executed.
Then, `ae 8b` is popped into `er14`. (that is, `8baeh`)
`5e b6 30 ??` corresponds to the address `0:b65eh`. The commands from `0:b65eh` to `0:b662h` are executed, effectively jumps to the start of the loop.

> There are a lot of `mov sp, er14; pop er14; pop pc` command sequences.
> Choose one that you find typing most easily.

Loops which modifies the stack

What is "modify the stack" and what is "not modify the stack"? This is pretty self-explanatory.

Note:

- A pop command only increases the stack pointer, does not change the value on the stack.
- A `push` command often modifies the stack, unless it's possible to prove that the stack value is always equal to the pushed value. That includes `push lr`.

So, to loop we should:

- When `sp = A`, and a `pop pc` command is executed, execute something that may modifies the stack.
- Return the stack to the original value.
- Execute a gadget with `pop er14; pop pc` and put `A-2` on the top of the stack.
- Execute a gadget with `mov sp, er14; pop er14; pop pc`.

Typically the stack is returned to the original value by calling the null-terminated string copy function on a 100-byte region that is not modified before the used stack content. As it's quite hard to determine which region is "not modified", this is often done by trial and error.

Example

TODO

TODO: Is there any unclear part (that you can't understand)?

---

from user202729