Imagine: pwning your favourite Nintendo DS game!

RomHack Camp 2026

by daisy

Who am I?

Hi everyone! Some info about me:

- `Vulnerability Researcher` @ `Bynario`
- Member of the `fibonhack` CTF team
- `Computer Engineering` undergrad @ UNIPI
- I like `software exploitation` and consoles architecture (NDS, PSP, Wii…)

Read more on my blog:

@daisyydaisy_

@daisydaisyyy

giulia-aloia-947555283

Who am I?

Hi everyone! Some info about me:

- Vulnerability Researcher @ Bynario
- Member of the fibonhack CTF team
- Computer Engineering undergrad @ UNIPI
- I like software exploitation and consoles architecture (NDS, PSP, Wii…)

Read more on my blog:

@daisyydaisy_

@daisydaisyyy

giulia-aloia-947555283

What's your favourite Nintendo DS game?

`Imagine: Figure Skating` is a Nintendo DS game developed by Spike

The goal is to improve your stats by doing weekly activities, then compete every month in figure skating competitions.

What's your favourite Nintendo DS game?

`Imagine: Figure Skating` is a Nintendo DS game developed by Spike

The goal is to improve your stats by doing weekly activities, then compete every month in figure skating competitions.

> When I was a child, I loved this game :)

Characters

- `3 different characters` to choose from
- You can save up to `2 different playthroughs`

How does the game handle the save data?

Each slot’s character data is `compressed` and then saved in the save file.

Instead of writing the full uncompressed game state, the game can write a smaller compressed version to save space.

How does the game handle the save data?

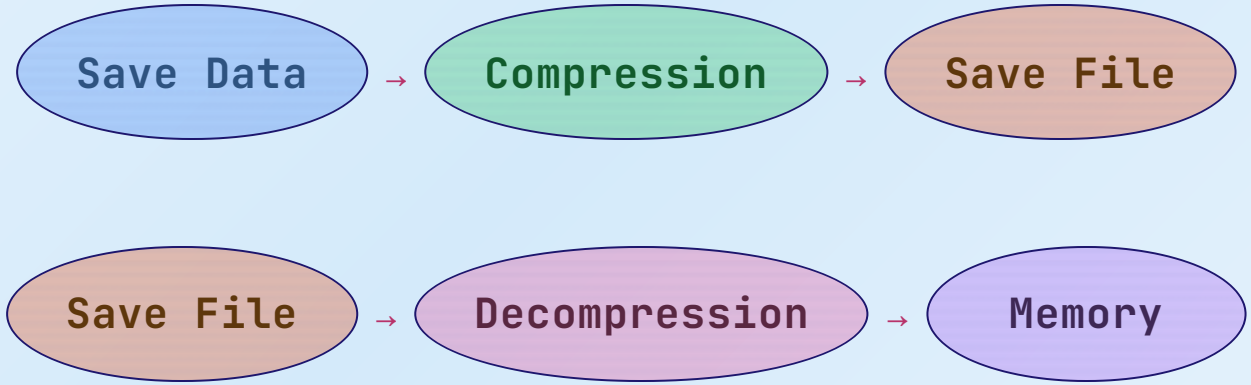
Each slot's character data is `compressed` and then saved in the save file.

Instead of writing the full uncompressed game state, the game can write a smaller compressed version to save space.

How does the game handle the save data?

Each slot's character data is `compressed` and then saved in the save file.

Instead of writing the full uncompressed game state, the game can write a smaller compressed version to save space.



Custom compression algorithms

Nintendo DS and GBA use compression algorithms, like LZ10 and LZ11.
Source: https://problemkaputt.de/gbatek-lz-decompression-functions.htm

```
  if typ=10h          ;LZ10 aka LZSS (BIOS SWI compatible)
    len=3
    len=len+[src]/10h, disp=001h+([src] AND 0Fh)*100h+[src+1], src=src+2
  elseif typ=11h      ;LZ11 (special extended format)
    if [src]/10h>1 then len=001h
    if [src]/10h<1 then len=011h+([src] AND 0Fh)*10h, src=src+1
    if [src]/10h=1 then len=111h+([src] AND 0Fh)*1000h+[src+1]*10h, src=src+2
    len=len+[src]/10h, disp=001h+([src] AND 0Fh)*100h+[src+1], src=src+2
  elseif typ=59h      ;YAZ0 (special extended format)
    len=[src]/10h, disp=001h+([src] AND 0Fh)*100h+[src+1], src=src+2
    if len=0 then len=[src]+12h, src=src+1, else len=len+02h
  endif
  for i=1 to len, [dst]=[dst-disp], dst=dst+1, next i
endif
```

Looking at the decompiled code:

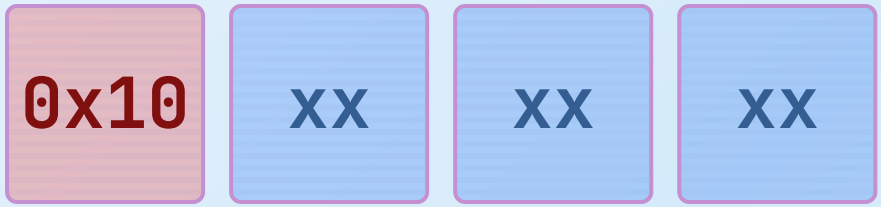
```
 // compress data and write it into the save file
base_offset = get_record_size(slotIndex);
save_succeed = false;
if (((10 < payload_len) && (slotTable[slotIndex * 8] != '\0')) &&
    (compr_size = lz_compress(dest + 1, payload_len, buffer), tmp = buffer, compr_size != 0)) {
  dest->magic = 0x7e;  // magic bytes -> valid slot
  dest->version = 0x1;
  dest->saveVersion = saveState[7];
  checksum_seed = base_offset + 7;
  dest->comprSize = (uint16_t)compr_size;
  checksum = calc_checksum(checksum_seed, buffer, compr_size);
  dest->checksum = checksum;
  compr_size = compr_size + 8;
  save_succeed = true;
  saved_slot_ptr = &dest->magic;
}
```

```
// in lz compress function
 *dst_buf = unc_size << 8 | 0x10; // writes 0x10 = compression type
 comp_size = 4;
 flag_ptr = dst_buf + 1;
 rem_size = unc_size;
 next_src = src_buf;
```

LZ10 compression algorithm

The compressed stream starts with a `header`:

Header layout



`0x10` = compression type (LZ10)

`xx xx xx` = size of decompressed data (24 bits)

After the header, the compressed stream is made of `blocks` containing 8 bits of data.

Each block starts with a `flag byte`.

Each bit tells the decompressor what to do:

Bit = 0

Copy next byte directly to output

Bit = 1

Back-reference follows: copy from output

After the header, the compressed stream is made of `blocks` containing 8 bits of data.

Each block starts with a `flag byte`.

Each bit tells the decompressor what to do:

Bit = 0

Copy next byte directly to output

Bit = 1

Back-reference follows: copy from output

Back-references reuse sequences already in output.

After the header, the compressed stream is made of `blocks` containing 8 bits of data.

Each block starts with a `flag byte`.

Each bit tells the decompressor what to do:

Bit = 0

Copy next byte directly to output

Bit = 1

Back-reference follows: copy from output

Back-references reuse sequences already in output.

Seems like an effective compression, but a small input can produce a much larger output!

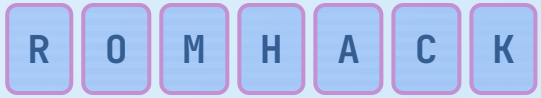
Backreferences

A back-reference is encoded using `2 bytes`:

```
length   = (first_byte >> 4) + 3    # range: 3..18
distance = (((first_byte & 0x0f) << 8) | second_byte) + 1 # range: 1..0x1000
source   = current_output_position - distance
```

LZ10 examples!

current output

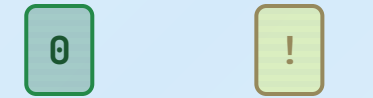


LZ10 examples!

current output

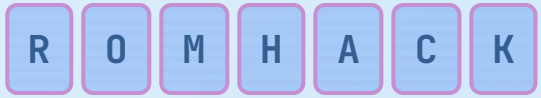
R O M H A C K

compressed input

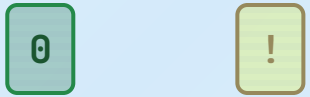


LZ10 examples!

current output

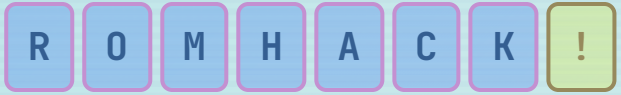


compressed input

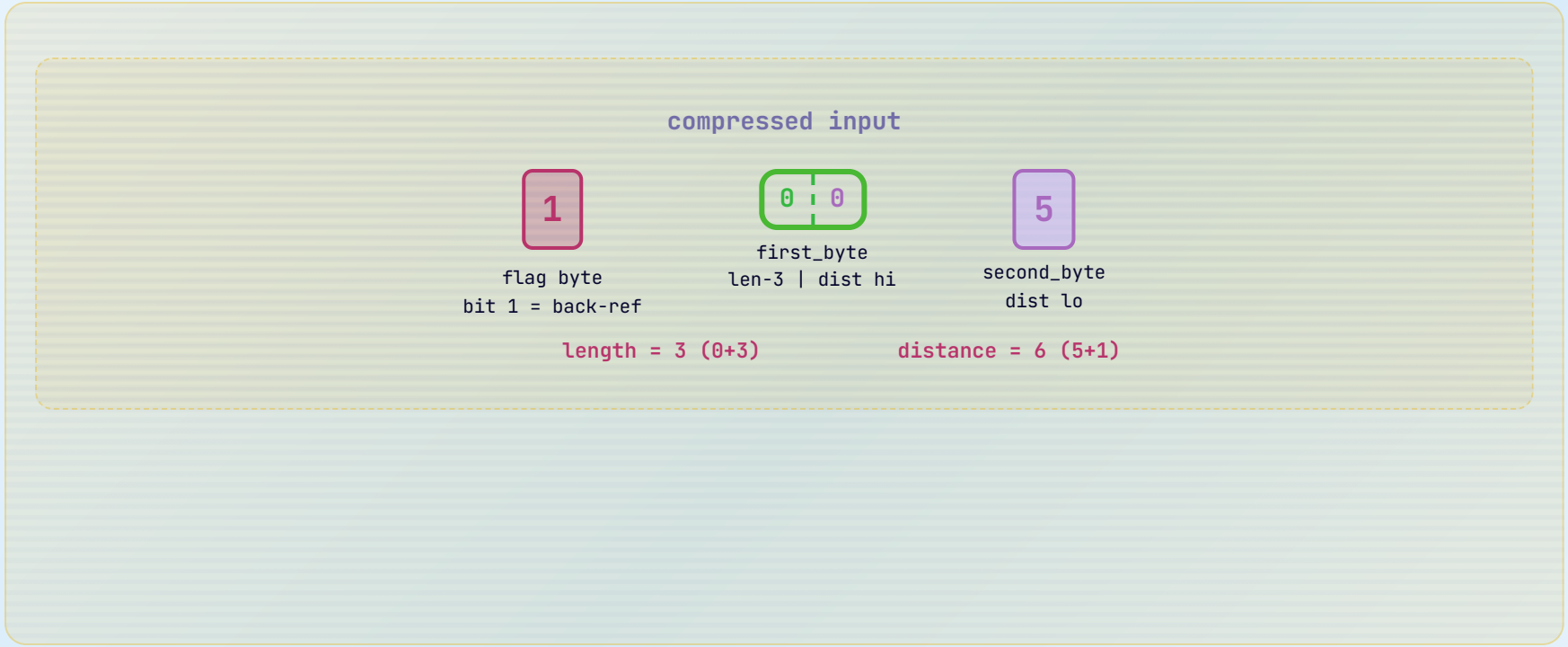


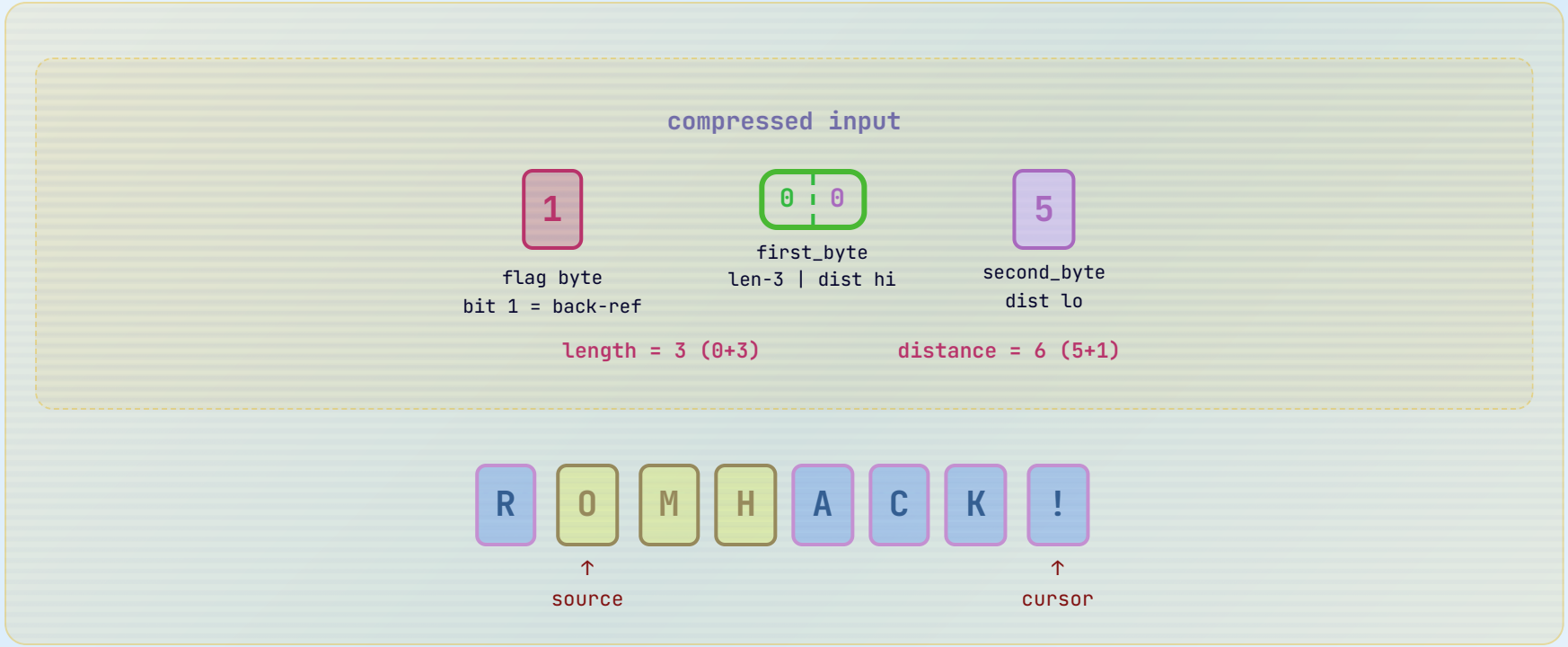
flag bit literal byte

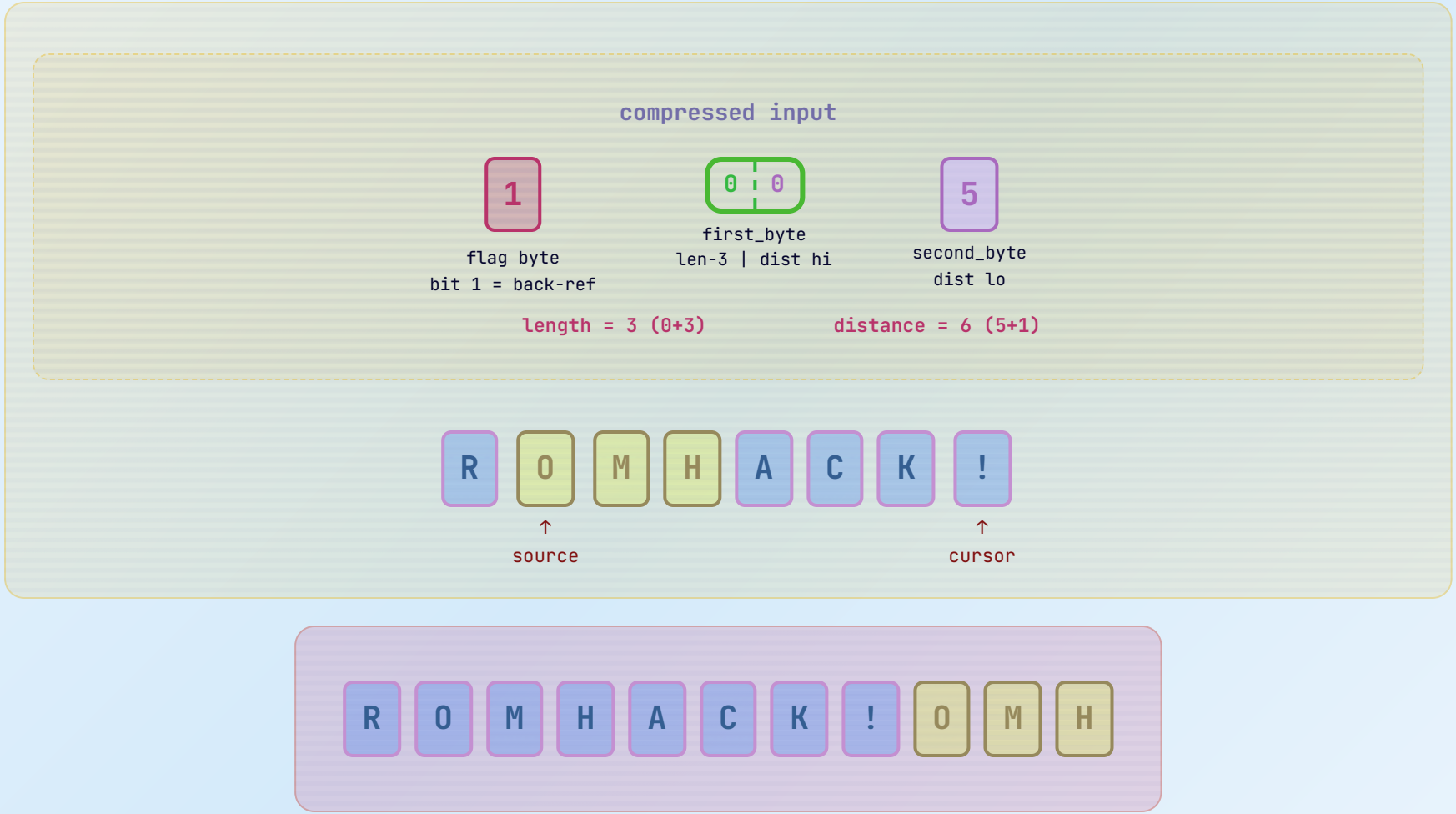
Flag bit = 0 → the next byte is copied literally on the current position



the byte "!" is appended


compressed input
1
flag byte
bit 1 = back-ref
0 : 0
first_byte
len-3 | dist hi
5
second_byte
dist lo
length = 3 (0+3)
distance = 6 (5+1)


compressed input
1
flag byte
bit 1 = back-ref
0 0
first_byte
len-3 | dist hi
5
second_byte
dist lo
length = 3 (0+3)
distance = 6 (5+1)
R O M H A C K !
source
cursor


compressed input
1
flag byte
bit 1 = back-ref
0 0
first_byte
len-3 | dist hi
5
second_byte
dist lo
length = 3 (0+3)
distance = 6 (5+1)
R O M H A C K !
source
cursor
R O M H A C K ! O M H

Looking at the decompress function

```
  else {
                /* flag = 1 -> copy from already existing output */
    length = ((int)(uint)(byte)*comp_nextdata >> 4) + 3;
    first_byte = *comp_nextdata;
    comp_data = (uint *)((int)comp_nextdata + 2);
    second_byte = *(byte *)((int)comp_nextdata + 1);
    n = -length;
    do {
      *out_buffer = out_buffer[-(((uint)second_byte | ((byte)first_byte & 0xf) << 8) + 1)];
                /* *out_buffer = out_buffer[-distance] */
      out_buffer = out_buffer + 1;
      remaining = length + -1;
      len_positive = 0 < length;
      length = remaining;
    } while (remaining != 0 && len_positive);
  }
  comp_pos = comp_pos + n;
  if ((int)comp_pos < 1) break;
  flag = flag << 1;
  comp_nextdata = comp_data;
  iVar1 = iVar1 + -1;
}
```

Woopsie, a bug!

Woopsie, a bug!

The requested distance is subtracted from the current output pointer.
But there is no check if it will be outside the output buffer.

Woopsie, a bug!

The requested distance is subtracted from the current output pointer.
But there is no check if it will be outside the output buffer.

Remember to check `distance ⩽ bytes_already_written` :)
Otherwise the source pointer could land outside the output buffer!

Leaking things

We can `copy data before the beginning of the decompression output buffer`:

Leaking things

We can `copy data before the beginning of the decompression output buffer`:

Insert a back reference with a distance bigger than the bytes written so far in the out buffer

`source = cur_output_ptr - distance` **distance > bytes written → source before buffer**

Leaking things

We can `copy data before the beginning of the decompression output buffer`:

Insert a back reference with a distance bigger than the bytes written so far in the out buffer

source = cur_output_ptr - distance

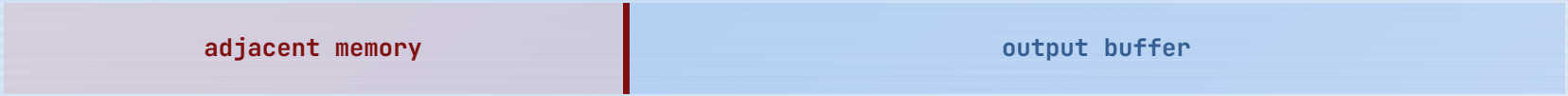
distance > bytes written → source before buffer

Leaking things

We can `copy data before the beginning of the decompression output buffer`:

Insert a back reference with a distance bigger than the bytes written so far in the out buffer

`source = cur_output_ptr - distance` **distance > bytes written → source before buffer**



We can leak adjacent memory by copying it into the output buffer!

Overwrite things (part 1)

When you start the game and click continue:
the UI shows both save slots *along with the player's name*.

We understood that:

1. The save's `CRC is validated`
2. Its data is `decompressed and copied` in the output buffer
3. The load screen `renders the save info`

Overwrite things (part 2)

Who renders the save info?

Let's look at the `function that shows the saves info` on the load screen:

```
 textbox_write(
  *(textBox + 0x24),           // textbox
  (char *)(rowObject->slot * 0x224 + loadSlotUi + 0x3a), // source string
  (ushort)uVar2, (ushort)arg3
);
```

Overwrite things (part 2)

Who renders the save info?

Let's look at the `function that shows the saves info` on the load screen:

```
textbox_write(
  *(textBox + 0x24),           // textbox
  (char *)(rowObject->slot * 0x224 + loadSlotUi + 0x3a), // source string
  (ushort)uVar2, (ushort)arg3
);
```

The source string comes straight from the `decompressed save data`

The optimized strcpy

```
 /* optimized strcpy - copies 4 bytes at once */
void string_copy(char *dest, char *source) {
  // ...
  source_word = *(uint32_t *)source;

  /* while there are no NULL bytes */
  while ((source_word + 0xfefefeff & ~source_word & 0x80808080) == 0) {
    *dest_ptr = source_word;
    source += 4;
    source_word = *(uint32_t *)source;
  }
  // ...
}
```

The optimized strcpy

```
 /* optimized strcpy - copies 4 bytes at once */
void string_copy(char *dest, char *source) {
  // ...
  source_word = *(uint32_t *)source;

  /* while there are no NULL bytes */
  while ((source_word + 0xfefefeff & ~source_word & 0x80808080) == 0) {
    *dest_ptr = source_word;
    source += 4;
    source_word = *(uint32_t *)source;
  }
  // ...
}
```

The stopping condition is only source byte == 0. If the profile name doesn't contain a null byte before the end of the textbox buffer, the copy continues into adjacent memory.

We can overflow the destination buffer and overwrite adjacent objects!

Load slot layout (part 1)

We need to understand where the save slot is loaded, maybe we can overwrite useful objects from there :)

With `gdb` we can see that the textbox buffer is located at:

```
(gdb) x/wx $r0+0x50
0x211870c:  0x022dbd60

(gdb) x/s 0x22dbd60
0x22dbd60:  "Daisy"  // that's the player's name!
```

If we inspect what's after that address:

```
(gdb) x/8wx 0x22dbf80
0x22dbf80:  0x02105cf8  0x02036524  0x0000003c  0x00000000
0x22dbf90:  0xffffffff  0x00000000  0x00000000  0x00012800
```

Load slot layout (part 1)

We need to understand where the save slot is loaded, maybe we can overwrite useful objects from there :)

With `gdb` we can see that the textbox buffer is located at:

```
(gdb) x/wx $r0+0x50
0x211870c:  0x022dbd60

(gdb) x/s 0x22dbd60
0x22dbd60:  "Daisy"  // that's the player's name!
```

If we inspect what's after that address:

```
(gdb) x/8wx 0x22dbf80
0x22dbf80:  0x02105cf8  0x02036524  0x0000003c  0x00000000
0x22dbf90:  0xffffffff  0x00000000  0x00000000  0x00012800
```

Load slot layout (part 2)

At `0x22dbf80+0x4` there is a `binary address` pointing to code.

```
x/8i 0x02036524
   0x2036524:     push     {r3, r4, r5, lr}
   0x2036528:     mov  r4, r1 ; r1 = event id
   0x203652c:     mov  r5, r0 ; r0 = obj pointer
   0x2036530:     cmp  r4, #1 ; checks event type
   0x2036534:     bne  0x20365d0 ; jumps if event type ≠ 1
   0x2036538:     ldr  r1, [r5, #20] ; access object field
```

Maybe it's a `UI object handler` ??

```
void HandleUiObjectEvent(uint eventId) { /* UI object event dispatcher */
  uiObject = *(int *)(DAT_0000ee3c + 8);
  if (uiObject == 0) return;
  do {
    (**(code **)(uiObject + 4))(uiObject, eventId);
    uiObject = *(int *)(uiObject + 0x58);
  } while (uiObject ≠ 0);
}
```

UI objects (part 1)

Each interactive element (buttons, text boxes, menu rows…):

- is represented as a `UI object`
- is manually assigned to the right handler function

→ `Active UI objects` are stored in a `singly-linked list`

`event` = UI notification sent by the engine (update, draw, confirm, cancel, selection change).

- the dispatcher broadcasts that event to every active UI object
- each object’s handler receives the event ID and decides whether to react or ignore it

UI objects (part 2)

For each event:

- the dispatcher `walks the list`,
- reads the function pointer at `object+4`,
- calls it with `r0 = object` and `r1 = eventId`,
- then `moves to the next object` through `object+0x58`.

```
 # Checking the linked list head:
(gdb) x/wx 0x0215CF44
0x215cf44:  0x022dbf80  // that's the address near our saved data!
```

UI objects (part 2)

For each event:

- the dispatcher `walks the list`,
- reads the function pointer at `object+4`,
- calls it with `r0 = object` and `r1 = eventId`,
- then `moves to the next object` through `object+0x58`.

```
 # Checking the linked list head:
(gdb) x/wx 0x0215CF44
0x215cf44:  0x022dbf80  // that's the address near our saved data!
```

Overwrite object+4 → redirect execution!

A simple exploit

We can build a simple exploit to `turn both screens red`:

A simple exploit

We can build a simple exploit to `turn both screens red`:

1. Read clean save → 2. Write payload → 3. Compress slots → 4. Write to the save

```
 # how to compress your payload
def compress_slot(save, slot_offset, plain_payload):
    save_version = u16le(save, slot_offset + 4)
    compressed_data = lz10(plain_payload, SLOT_SIZE - RECORD_HEADER_SIZE)

    record = bytearray(b"\xFF" * SLOT_SIZE)
    record[0] = MAGIC # 0x7E
    record[1] = 1
    record[2:4] = p16(len(compressed_data))
    record[4:6] = p16((save_version + 1) & 0xFFFF)
    record[6:8] = p16(crc16(compressed_data, SLOT_SIZE + 7))
    record[RECORD_HEADER_SIZE:RECORD_HEADER_SIZE + len(compressed_data)] = compressed_data
    save[slot_offset:slot_offset + SLOT_SIZE] = record
```

A simple exploit

We can build a simple exploit to `turn both screens red`:

1. Read clean save → 2. Write payload → 3. Compress slots → 4. Write to the save

To control the screen we need to know how the NDS renders graphics :)

```
record = bytearray(b"\xFF" * SLOT_SIZE)
record[0] = MAGIC # 0x7E
record[1] = 1
record[2:4] = p16(len(compressed_data))
record[4:6] = p16((save_version + 1) & 0xFFFF)
record[6:8] = p16(crc16(compressed_data, SLOT_SIZE + 7))
record[RECORD_HEADER_SIZE:RECORD_HEADER_SIZE + len(compressed_data)] = compressed_data
save[slot_offset:slot_offset + SLOT_SIZE] = record
```

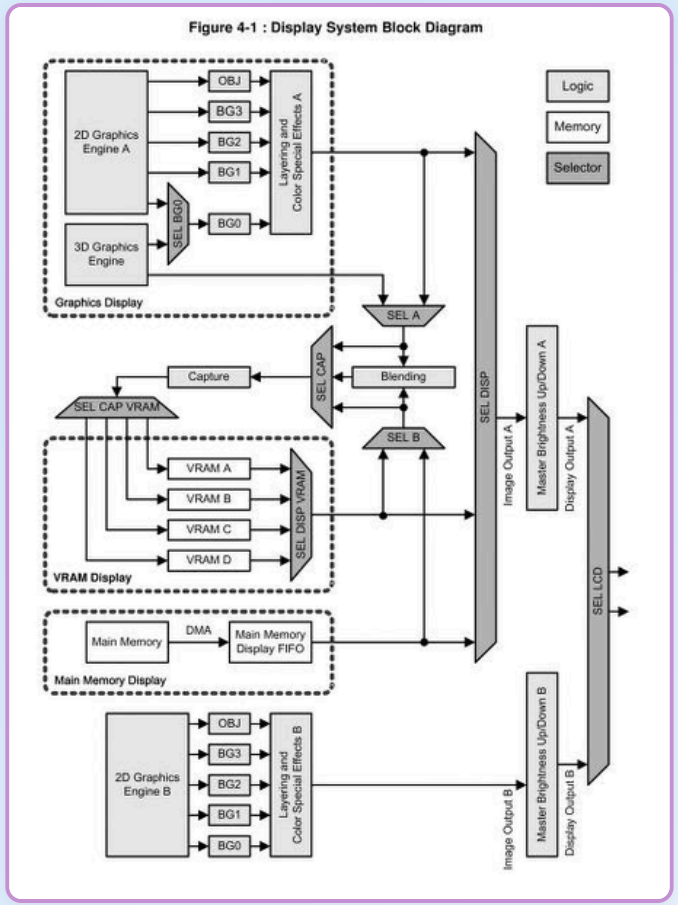
NDS graphics engines

Nintendo DS has `two independent graphics engines`:

- `Engine A (Main)`: supports 2D and 3D rendering
- `Engine B (Sub)`: only supports 2D rendering

Each engine can be swapped between the two screens by setting bit 15 of the `POWCNT1` register.

source: NitroDev Docs



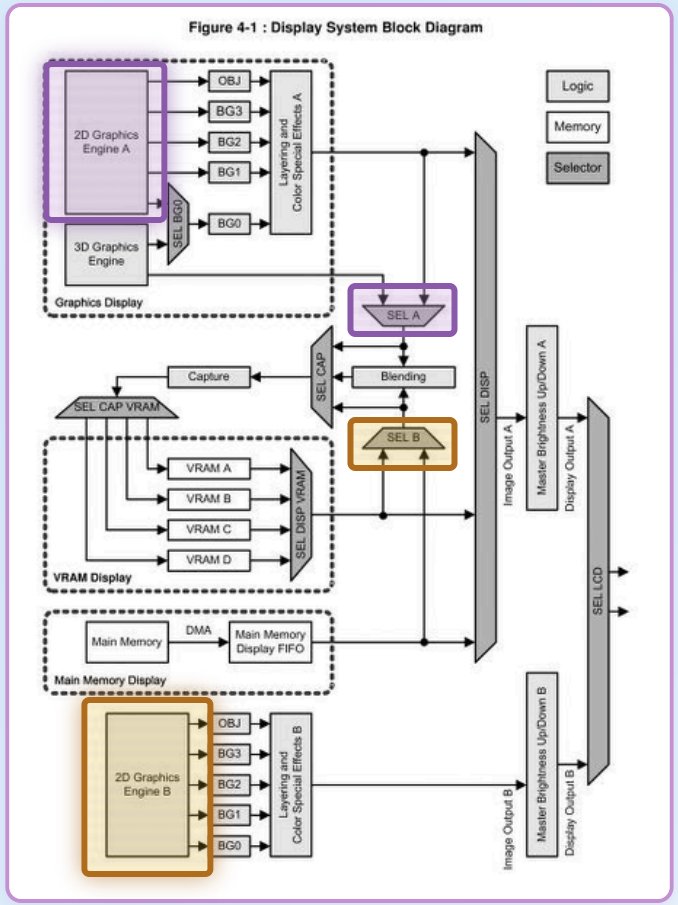
NDS graphics engines

Nintendo DS has `two independent graphics engines`:

- `Engine A (Main)`: supports 2D and 3D rendering
- `Engine B (Sub)`: only supports 2D rendering

Each engine can be swapped between the two screens by setting bit 15 of the `POWCNT1` register.

source: NitroDev Docs



Graphic modes

Each pixel’s color can be resolved in two different ways, depending on the active mode:

The active mode is set with the `DISPCNT` register's `BG Mode` field (bits 0-2, values 0-6)

tile-based mode = VRAM stores an *index*, color comes from the palette

direct bitmap mode = VRAM stores the *color itself*

Graphic modes

Each pixel's color can be resolved in two different ways, depending on the active mode:

The active mode is set with the DISPCNT register's BG Mode field (bits 0-2, values 0-6)

We don't know which mode is active:
so our shellcode writes red to
Palette RAM *and* VRAM

raw BGR555 in VRAM → 15-bit color

direct bitmap mode = VRAM stores the *color itself*

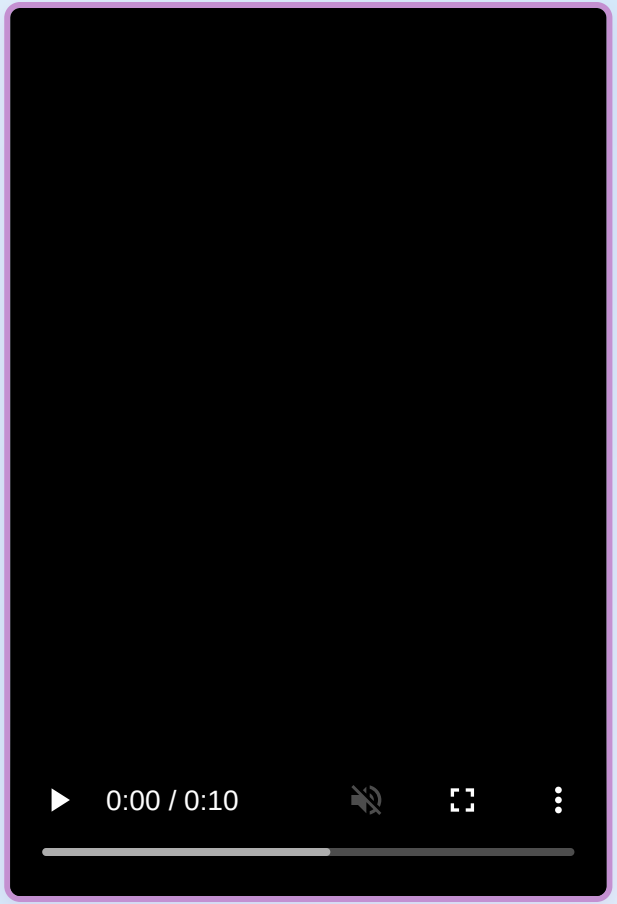
Shellcode: fill the RAM palette

```
@ set the red color (BGR555 format = 0x001F)
mov     r1, #0x1F
orr     r1, r1, r1, lsl #16 ; red in high half -> 2 pixels at once

@ fill RAM palette (2 KB)
mov     r0, #0x05000000 ; RAM palette base
mov     r2, #0x200 ; loop counter = 512

fill_palette:
  str     r1, [r0], #4 ; write red, advance ptr
  subs    r2, r2, #1
  bne     fill_palette
```

Shellcode: fill both VRAMs

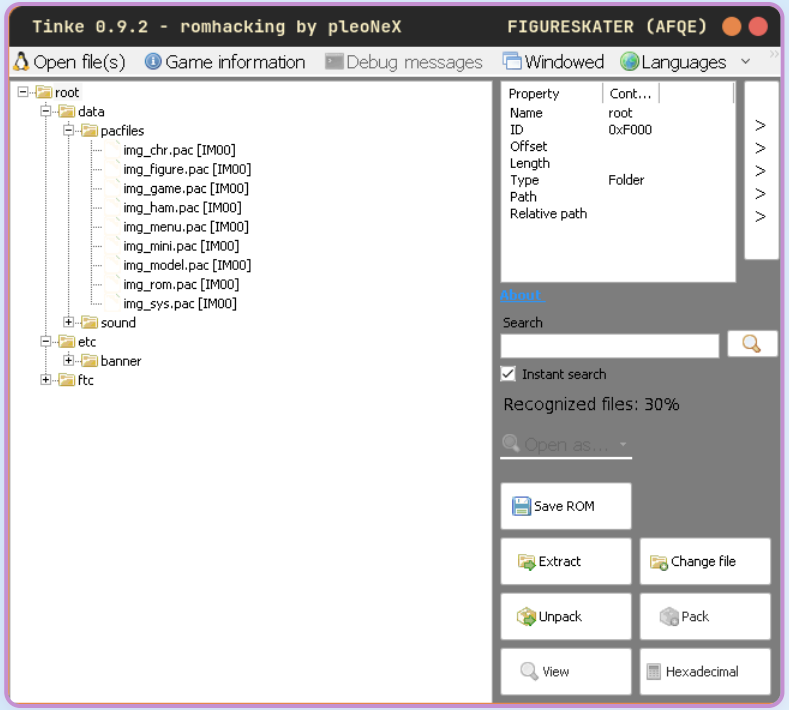
```
mov     r0, #0x06200000 ; VRAM for Engine B
mov     r2, #0x6000 ; 96 KB
loop_B:
  str     r1, [r0], #4
  subs    r2, r2, #1
  bne     loop_B

mov     r0, #0x06000000 ; VRAM for Engine A
mov     r2, #0x6000
loop_A:
  str     r1, [r0], #4
  subs    r2, r2, #1
  bne     loop_A

@ freeze the game
freeze:
  b freeze
```

This works!



Styling up :)

There are many accessories in this game to style your character.
Let's look at how they are handled. To extract the game's files, I use `Tinke`.



About game assets

Compressed assets are stored in `pac files` (a custom archive format):
a tiny header (magic + file count) followed by an offset table.
Assets are addressed by *index*.

- Assets are compressed using `LZ10` (the *same* algorithm used for the save files)
- Decompressed models are in the BMD0 format (`.nsbmd` file)

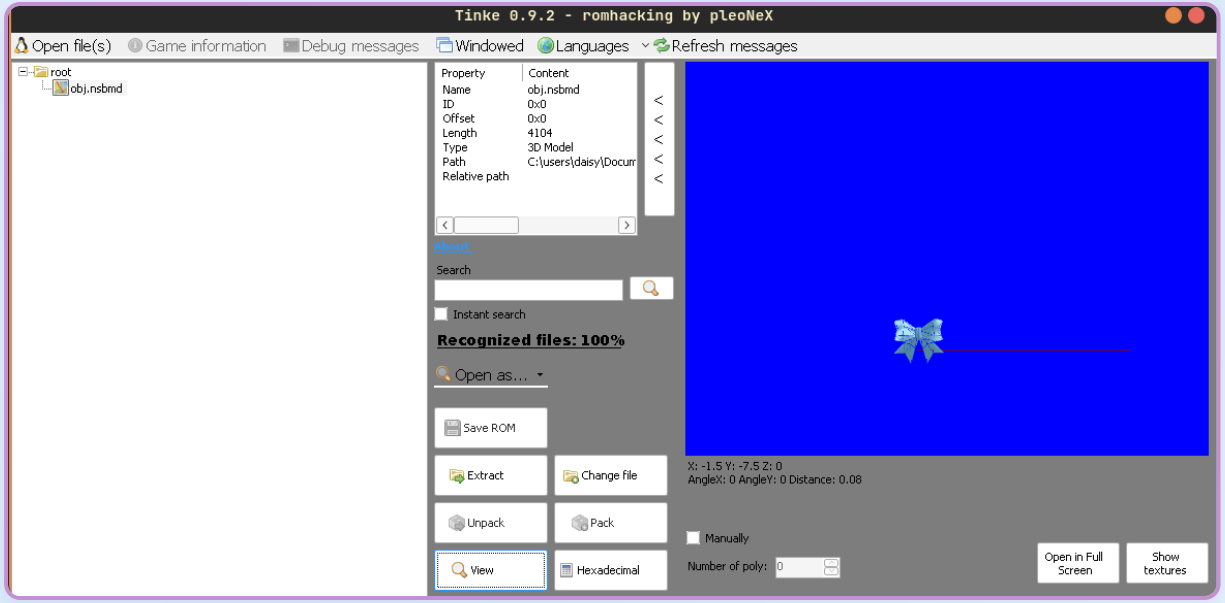
Interesting assets can be found in `img_figure.pac`

Extracting accessories

```
 # List accessories table
import struct, re

PAC = "./data/pacfiles/img_figure.pac"

with open(PAC, "rb") as f:
    data = f.read()

n = struct.unpack_from("<I", data, 4)[0]
offsets = [struct.unpack_from("<I", data, 8 + 4*i)[0]
           for i in range(min(n, (len(data)-8)//4))]

for idx in range(len(offsets)):
    chunk = data[offsets[idx]:offsets[idx+1]]
    if not chunk or chunk[0] ≠ 0x10: continue
    model = lz10_decompress(chunk)
    if model[:4] ≠ b"BMD0": continue
```

Extraction result

Each accessory has its own BMD0 model and a specific index in the PAC file



Asset `blue_R`, index 551

PAC file content

Some accessories are composed of multiple parts (left/right leg, left/right arm):

```
562  0x232  flower_1
563  0x233  gun
564  0x234  hamster
565  0x235  hat
566  0x236  heart
567  0x237  helmet
```

```
578  0x242  LL_heart      534  0x216  AL_heart
579  0x243  LL_skull      535  0x217  AL_skull
580  0x244  LL_star       536  0x218  AL_star
581  0x245  LR_heart      539  0x21B  AR_heart
582  0x246  LR_skull      540  0x21C  AR_skull
583  0x247  LR_star       541  0x21D  AR_star
```

`LL` / `LR` = left/right leg

`AL` / `AR` = left/right arm

Preview accessory table

The `preview accessory table` pointer (pointing to `0x0210AB88`) is used to decide which 3D model asset is loaded when the game previews a selected accessory.

```
                    ptr_accessory_preview_table_0210…  XREF[2]:  LoadAvatarAccessoryPre…
                    ptr_AccessoryPreviewTable                     LoadAvatarAccessoryPre…
000408cc 88 ab 10 02    undef…  0210AB88h
```

When the game needs to `display an accessory`:

- it reads this table to get the asset id,
- then calls `loadModelById` which receives the asset id
- extracts the file index and chunk index

Accessory preview loading

This function finds the chosen accessory in the pac file and loads the model preview:

```
void LoadAccessoryPreview(int previewObject, int equipIndex, int assetPart) {
  *(previewObject + 900) = equipIndex;
  if (assetPart == 0) {
    loadModelById(previewObject,
      *(uint)((equipIndex + 0x23) * 0x2c + ptr_AccessoryPreviewTable + 0x1c),
      1, equipIndex + 0x23);
  } else {
    packedAssetId = *(uint)((equipIndex + 0x23) * 0x2c + ptr_AccessoryPreviewTable + 0x28);
    if (packedAssetId != 0) { // accessory with multiple parts
      loadModelById(previewObject, packedAssetId, 1, equipIndex + 0x23);
    }
  }
}
```

Example: `packedAssetId = 0x00060233`

`0x0006` → file index

`0x0233` → accessory index

- If `the model isn't already cached`, it allocates memory, reads the compressed data from the pac file, and caches the result.
- The `compressed chunk is decompressed, then parsed as a` `BMD0` `model`.
- The parser `extracts vertices, normals, UV coordinates and calculates the bounding box` for camera positioning.
- The resulting 3D model is then handed to the `NDS GPU renderer for display`.

Where is the equipped accessory stored?

A player can have only `1 equipped accessory` at once.

The player data is loaded from the save file and copied at `0x0215E86C`:

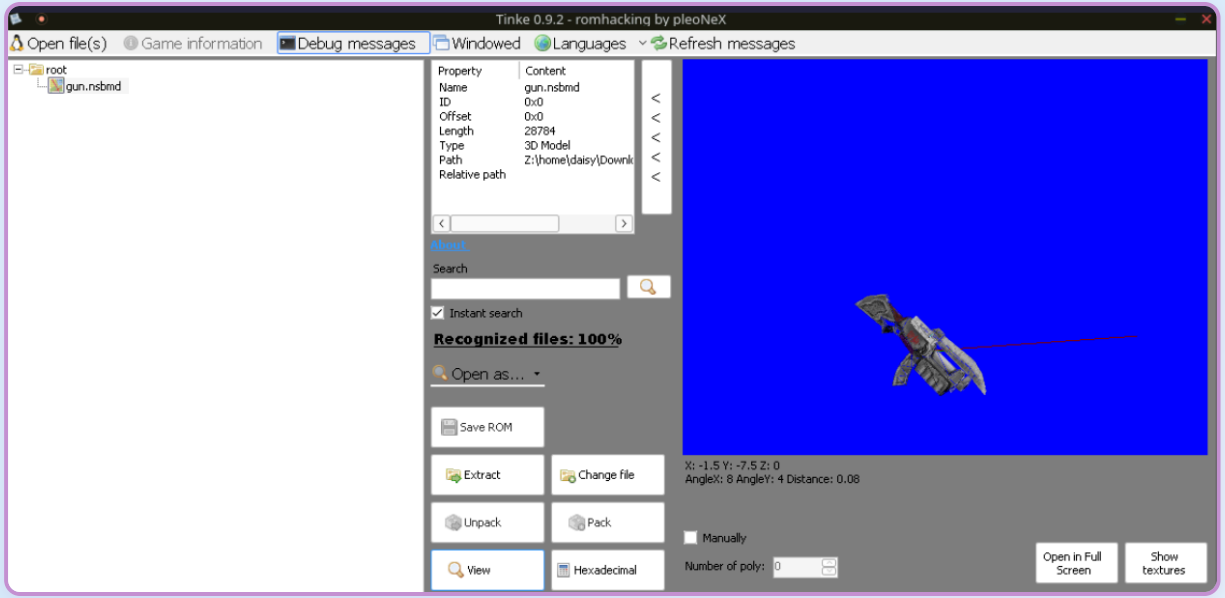
```
void LoadEquippedPreview(int param_1, int param_2) {
  // ...
  previewObject = calloc(0x390);
  // loads accessory → at characterData + 0x50!
  LoadAvatarAccessoryPreviewAsset(previewObject, *(ptrCharacterData + 0x50) + -2, 0);
  category = *(AccessoryCatTable + *(profileData + 0x50));
  // if 2-part accessory, also loads 2nd part
}
```

Current equipped accessory is at `player data + 0x50`

Another exploit idea

I was looking at game assets just for fun when I found this one that’s quite strange:

By dumping the accessory preview entries, no normal shop entry referenced the gun asset `0x00060233`, so it’s `not used` in the game!



Exploit: view the gun accessory!

1. Overflow slot 1 → 2. Overwrite handler → 3. Execute shellcode

The shellcode:

1. Patches the accessory table replacing an existing asset index with the gun index
2. Returns to the normal game flow

```
source+0x000..0x21F  → padding
source+0x220..0x223  → object+0x00 = 0x02105CF8
source+0x224..0x227  → object+0x04 = 0x022CAE28 ← our shellcode!
source+0x228..0x22B  → object+0x08 = 0x00000000
```

1. Load screen → 2. Select slot → 3. Map scene → 4. Navigate to shop →
5. Select Heart Elbow → 6. Gun rendering

First handler (part 1)

```
@ repair the object
ldr  r0, L_ui_object ; 0x022DBF80
ldr  r1, L_normal_word0 ; 0x02105CF8   ; the "normal" word for +0x00
str  r1, [r0] ; restore object+0x00

ldr  r1, L_second_handler
str  r1, [r0, #4] ; object+0x04 = second handler
```

The dispatcher is still walking the UI object list when it calls **our first handler**: patching the accessory table at this point would break the game.

- restore `object+0x00`: the object stays valid after the handler returns
- edit `object+0x04 = second handler`

On the next UI tick, the game calls `object→handler` again, this time it runs the **second handler**.

First handler (part 2)

The load menu

On the load menu, the game reads a state value from the load-controller and decides what happens next.

1. `0x65` = state the menu is in when our handler runs
2. `0x66` = the save starts loading
3. `0x67` = poll: wait until the load is done
4. `0x68` = build the map scene
5. `0x3e8` = back to the load screen

```
if (cur_substate < 0x67) {
  if (0x65 < cur_substate) {
    if (cur_substate != 0x66) {
      return;
    }
                    /* substate 0x66 -> controller commits to load a specific slot */
    state = loadRowState->active;
                    /* wait */
    if (state->loadReady == 0) {
      return;
    }
                    /* error */
    if (state->loadReady == 2) {
      loadController->retryTimer = 0x5a;
      loadController->substate = 0x6e;
      rowState->active->loadReady = 0;
      rowState->active->selectedSlot = -1;
      return;
    }
                    /* loadReady = 1 -> slot is selected and ready */
    *(byte *)(loadMenuFlags + 7) = *(byte *)(loadMenuFlags + 7) | 0x30;
    SaveAsync_SetRequest(state->selectedSlot + 1,1,0);
    sceneSetup(parentController->scene,DAT_00035064);
    notifyControllerState(loadController,1);
                    /* = 0x67 */
    loadController->substate = loadController->substate + 1;
    return;
  }
```

The first handler fakes the player's input instead of waiting for a button press:

`selectedSlot = 1` → slot 1 was chosen

`loadReady = 1` → the load was triggered

`loadResultB = 3` → result expected to load the game screen

```
 @ prepare load-controller fields
ldr  r0, L_load_row_state ; 0x0215E798
ldr  r0, [r0]
mov  r1, #1
str  r1, [r0, #0x28] ; selectedSlot = 1
str  r1, [r0, #0x490] ; loadReady = 1
mov  r1, #3
str  r1, [r0, #0x494] ; loadResultB = 3
```

Second handler

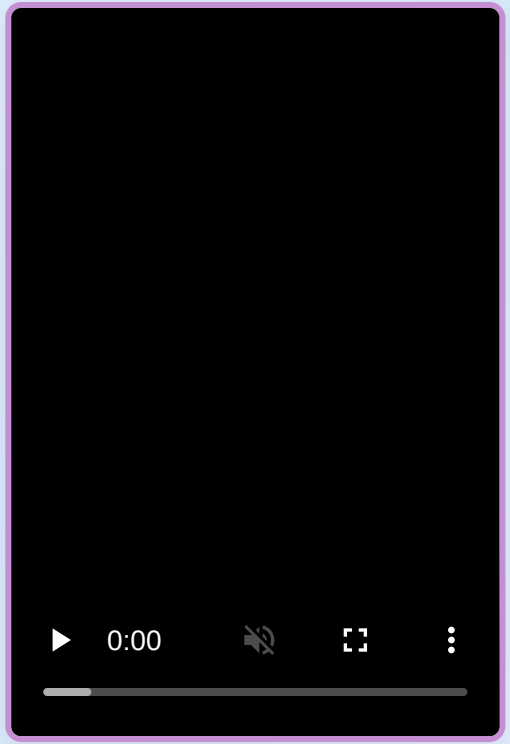
1. `Patch the accessory table`
2. `Resume the game` and it continues to the map screen using the normal routine

```
 @ patch heart elbow → gun
ldr  r1, L_gun_asset ; 0x00060233
ldr  r0, L_heart_elbow_asset_a ; 0x0210BBCC (was 0x00060216)
str  r1, [r0] ; entry A = gun
ldr  r0, L_heart_elbow_asset_b ; 0x0210BBD8 (was 0x0006021B)
str  r1, [r0] ; entry B = gun

@ resume the normal flow
blx  0x02086E68 ; parent scene dispatcher → continue the scene normally
blx  0x0201771C ; destroy fade object → close the screen transition
blx  0x02034454 ; load-controller cleanup → let the game clean up
```

Result

`0x0210BBCC = 0x00060216` → `0x00060233`

`0x0210BBD8 = 0x0006021B` → `0x00060233`



Extra: Action Replay (part 1)

Action Replay codes for the DS are pairs of 32-bit hex words:

- `code type`: write size
- `address`: offset from RAM base address. real address = `address | 0x02000000`
- `value`: the data to write

Code type	Write size
0	32-bit write
1	16-bit write
2	8-bit write

Extra: Action Replay (part 2)

Those codes just write a value into a RAM address.

It works because DS flashcarts intercept and patch memory while the game is running.

Example: get a lot of money :)

`1215E8D0 00002706`

`first hex digit = 1` → `16-bit write`

`0x15E8D0 | 0x02000000` → `0x0215E8D0` → money field

`0x2706` → `9990` coins

offset from player data (starting at `0x0215E86C`) → 0x64

Extra: why not just use Action Replay?

It is a cheat device and emulator feature that runs **outside the game** and changes RAM while the game is already running.

Action Replay

Cheat engine already has RAM write access

`address`

`+`

`value`

↓

`patched memory`

Our exploit

The game only receives a valid save file

`.sav`

↓

`parser`

↓

`overflow`

↓

`ARM9 code execution`

The end :)

Thank you!

Imagine: pwning your favourite Nintendo DS game!

ROMHACK CAMP 2026

𝕏 @daisyydaisy_

@daisydaisyyy