SHAttered Dreams

Adventures in BootROM Land

By Joshua Hill
(@p0sixninja)

Disclaimer

- Nothing in this talk should imply that I was responsible these BootROM exploits.
- Special thanks to my wonderful friend @pod2g for staying up late at night and arguing with me over why things were crashing.
- This is not my story, this is our story. You should be up here on stage with me now.

Introduction

- Who am I?
- What have I done?
- What's going to be covered?

Who am I?

- Joshua Hill (@p0sixninja)
- Experienced iOS Jailbreaker
- Self-taught Developer and Hacker

Accomplishments

- Worked with Chronic-Dev Team for 4 years. Currently independent researcher.
- Chief Architect behind the GreenPois0n and Absinthe jailbreaks.
- Been reversing iOS BootROM since 2008.
- Stole these slides from other presentations.

Agenda

- What is a BootROM
- How to dump BootROM
- BootROM walk-through
- Past BootROM exploits
- Exploitation methods

Terminology

- SROM - SecureROM
- SRAM - SecureRAM
- MMIO - Memory Mapped I/O
- MIU - Memory Interface Unit
- MMU - Memory Management Unit
- DFU - Device Firmware Update

Terminology (cont.)

- BSS - Incorrect term we used to describe the DATA section.
- SHSH - Secure Hash used for image validation
- LLB - Low Level Bootloader
- IRQ - Interrupt Request
- VIC - Vector Interrupt Controller

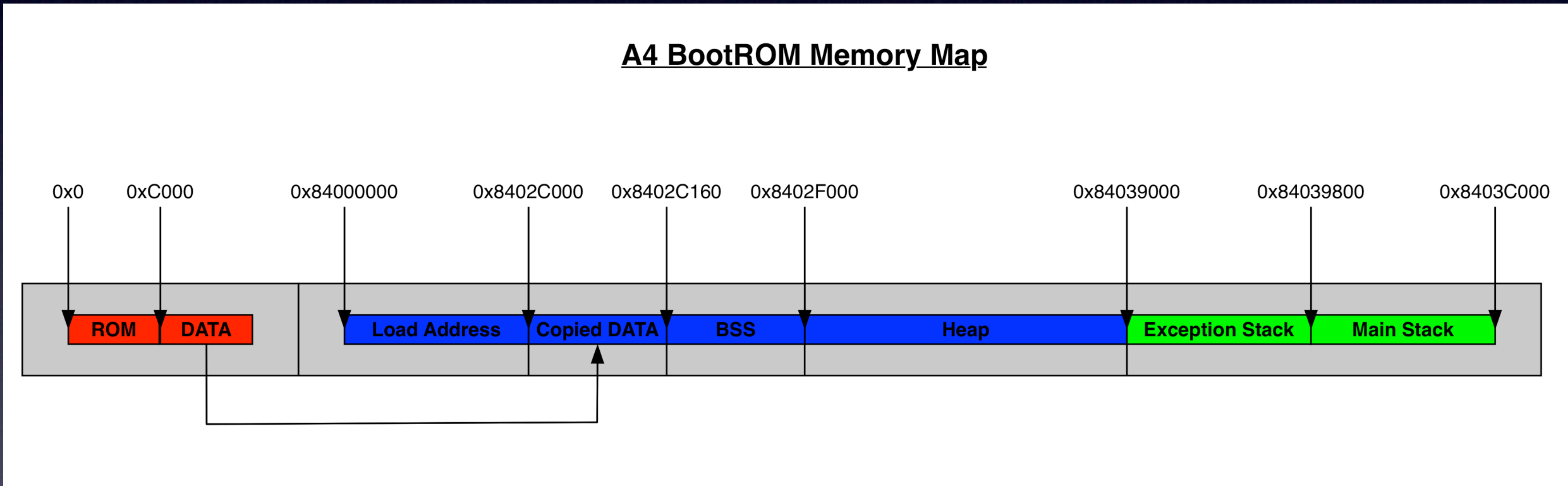
What is BootROM

- Boot-up Procedure
- Memory Mappings
- Device Changes

Boot-up Procedures

- Automatically mapped at 0x0 when powered on.
- Allocates a small section of SRAM for dynamic data.
- Chooses boot method or defaults back to DFU mode.

Memory Mappings



Device Changes

- Address of SRAM has changed over the years but always same address as LLB and iBSS.
- iPod2g was 0x22000000, A4 chips was 0x84000000
- Size of load address changed from 0x24000 bytes to 0x2C000 bytes after A4

How to Dump BootROM

- Dump from iBoot.
- Dump from Kernel.
- Exposing with MIU.

Dumping From Physical Memory

- This technique first discovered by @pod2g to dump BootROM from iBoot.
- No clue why it was worked at the time.
- Discovered later BootROM was actually remapping itself for some reason.

BootROM Randomly Remapping Itself

```
ROM:000044FE 01C          MOVS    R1, 0xBF000000
ROM:00004502 01C          MOVS    R0, R1
ROM:00004504 01C          MOVS    R2, #1
ROM:00004506 01C          MOVS    R3, #1
ROM:00004508 01C          STR     R6, [SP,#0x1C+var_1C]
ROM:0000450A 01C          STR     R4, [SP,#0x1C+var_18]
ROM:0000450C 01C          BL      mmu_map_addr
```

Dumping from iBoot With Cyanide

```
 hexdump 0xBF000200 0x40
xbf000200: 53 65 63 75 72 65 52 4f 4d 20 66 6f 72 20 73 35 SecureROM for s5
xbf000210: 6c 38 39 33 30 78 73 69 2c 20 43 6f 70 79 72 69 l8930xsi, Copyri
xbf000220: 67 68 74 20 32 30 30 39 2c 20 41 70 70 6c 65 20 ght 2009, Apple
xbf000230: 49 6e 63 2e 00 00 00 00 00 00 00 00 00 00 00 00 Inc.............
```

Dumping From Kernel Payload

```
id hook() {
  IOLog("p0sixninja is in da house!!\n");
  void* mem = IOMemoryDescriptor(0xBF000200, 0x40, 3);
  IOLog("mem = 0x%08x\n", mem);
  void* map = IOMemoryDescriptor_map(mem, 3);
  IOLog("map = 0x%08x\n", map);
  unsigned int* va = IOMemoryMap_getVirtualAddress(map);
  IOLog("va = 0x%08x\n", va);
  IOHexdump(va, 0x40);
```

Output From Kernel Payload

```
0sixninja is in da house!!
em = 0x89878930
ap = 0x895a95d8
a = 0xd3e0b200
xd3e0b200: 53 65 63 75 72 65 52 4f 4d 20 66 6f 72 20 73 35 SecureROM for s5
xd3e0b210: 6c 38 39 33 30 78 73 69 2c 20 43 6f 70 79 72 69 l8930xsi, Copyri
xd3e0b220: 67 68 74 20 32 30 30 39 2c 20 41 70 70 6c 65 20 ght 2009, Apple
xd3e0b230: 49 6e 63 2e 00 00 00 00 00 00 00 00 00 00 00 00 Inc.............
```

Exposing via MIU

- Trick discovered by @planetbeing I believe.
- By changing the value of MIU register in ARM-IO MMIO BootROM would magically appear back at 0x0.
- This allowed the first BootROM to be dumped.

BootROM Dumper

- Created by @pod2g using the SHAtter BootROM exploit.
- Place device in DFU mode and just let it run.
- https://github.com/Chronic-Dev/Bootrom-Dumper

A5 BootROM

- None of these tricks appear to work on A5 processor anymore.
- This would be the first step to exploiting any crashes discovered in new BootROM.
- Any hardware guys out there wanna take a stab at it?

BootROM Walk-through

- Start-Up
- Main Function
- DFU Mode
- Image Validation

Start-Up

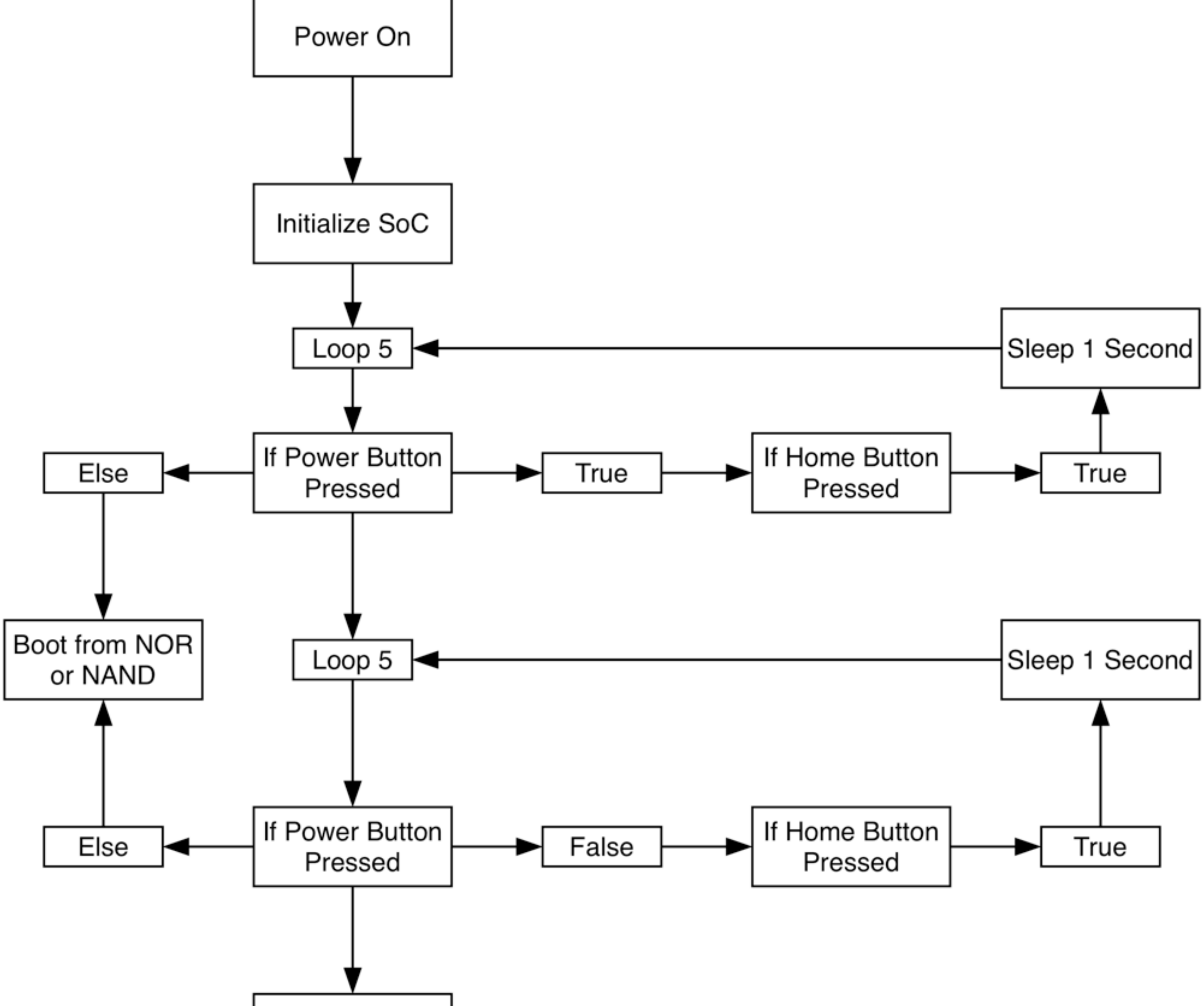
- Checks to ensure it's running at 0x0
- Copies DATA section from SROM to SRAM
- Clears memory where heap will be located
- Sets up exception stack and main stack
- Jumps to main function

Example From A4

- 0x160 bytes of data located at 0xC000 is copied to 0x8402C000
- Memory from 0x8402C160 to 0x84039000 is cleared for heap
- Exception stacks set to 0x84039800 and main stack set to 0x8403C000

Main Function

- Split into 2 different parts.
- First part checks which buttons are being held down the check for DFU boot.
- Second part checks boot method and proceeds to load and validate image.


Power On
Initialize SoC
Loop 5
Sleep 1 Second
Else
If Power Button Pressed
True
If Home Button Pressed
True
Boot from NOR or NAND
Loop 5
Sleep 1 Second
Else
If Power Button Pressed
False
If Home Button Pressed
True

Normal Boot Method

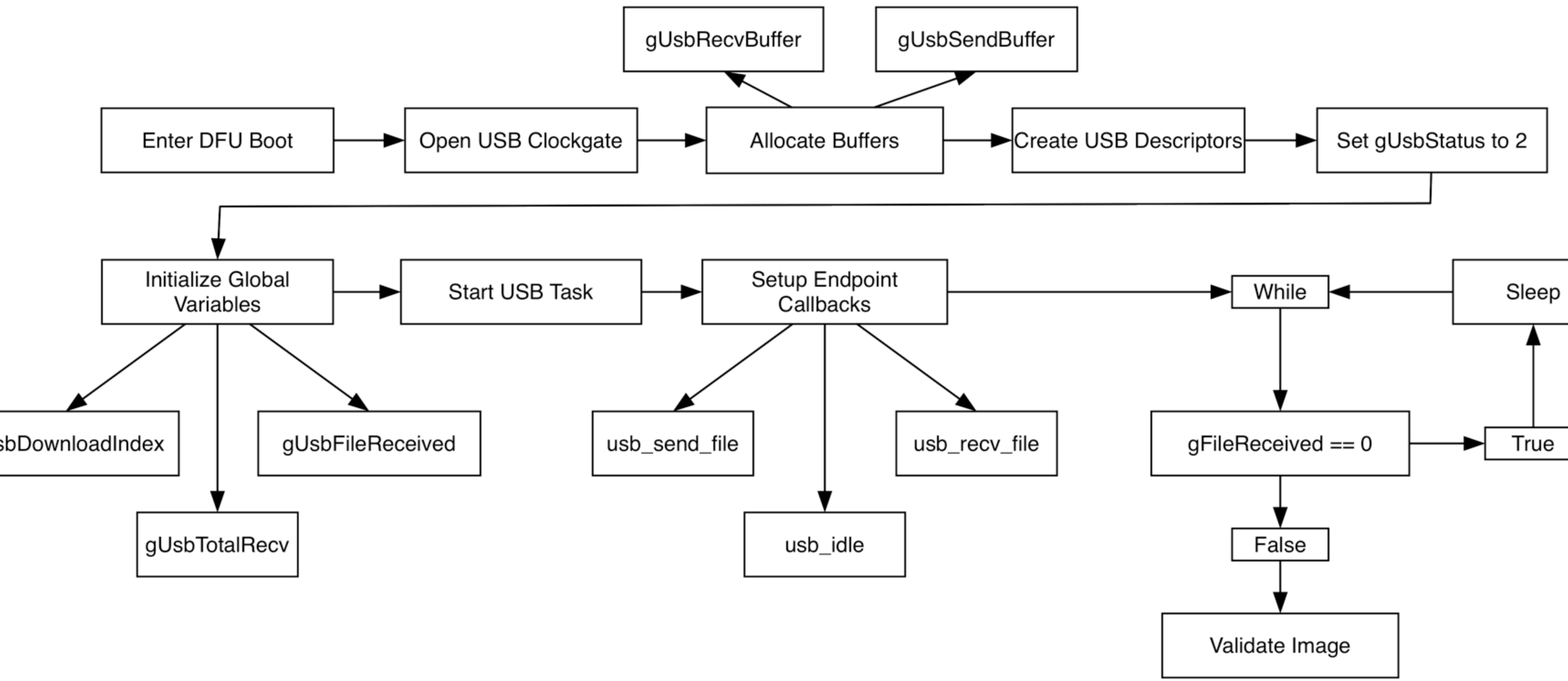
- In iPod2g images were loaded from NOR flash chip.
- Modern devices all load images from NAND flash.
- If loading from NOR/NAND fails, BootROM defaults into DFU mode.

DFU Mode

- USB code receives most of the changes in each BootROM revision.
- Very buggy portion of code!
- 3 out of 5 exploits discovered were in this portion of BootROM

DFU Initialization

- Allocates memory for send and receive buffers.
- Resets global variables to known state.
- Sets up USB descriptors, interfaces, and registers callbacks for endpoints.
- Enters infinite loop waiting for global "file received" variable to be set.


gUsbRecvBuffer
gUsbSendBuffer
Enter DFU Boot
Open USB Clockgate
Allocate Buffers
Create USB Descriptors
Set gUsbStatus to 2
Initialize Global Variables
Start USB Task
Setup Endpoint Callbacks
While
Sleep
sbDownloadIndex
gUsbFileReceived
gUsbTotalRecv
usb_send_file
usb_recv_file
usb_idle
gFileReceived == 0
True
False
Validate Image

USB Task

- When USB packet sent, device triggers an IRQ.
- Interrupt handler looked up in VIC table.
- Simple requests are handled in USB interrupt handler.
- Other requests are queued up to be handled later.

Control Request Packets

- 0x21, 1 - Send Data
- 0xA1, 2 - Recv Data
- 0xA1, 3 - Get Status
- 0x21, 4 - Reset Counters
- 0xA1, 5 - Get State

Ending USB Task

- Image validation starts whenever the global "file received" variable has been set.
- This can be caused by sending 1 empty "Send Data" packet, and 3 "Get Status" packets followed by a USB reset.
- Or when the maximum send or receive size has been reached.

Image Validation

- Image Descriptor
- Signature Check
- Device Check
- Image Decryption

Image Descriptor

- Memz structure holds information about image in memory.
- Information in Img3 header compared with information in Memz structure.
- Includes flags describing what kind of image, if it was loaded from NAND or DFU.

Signature Check

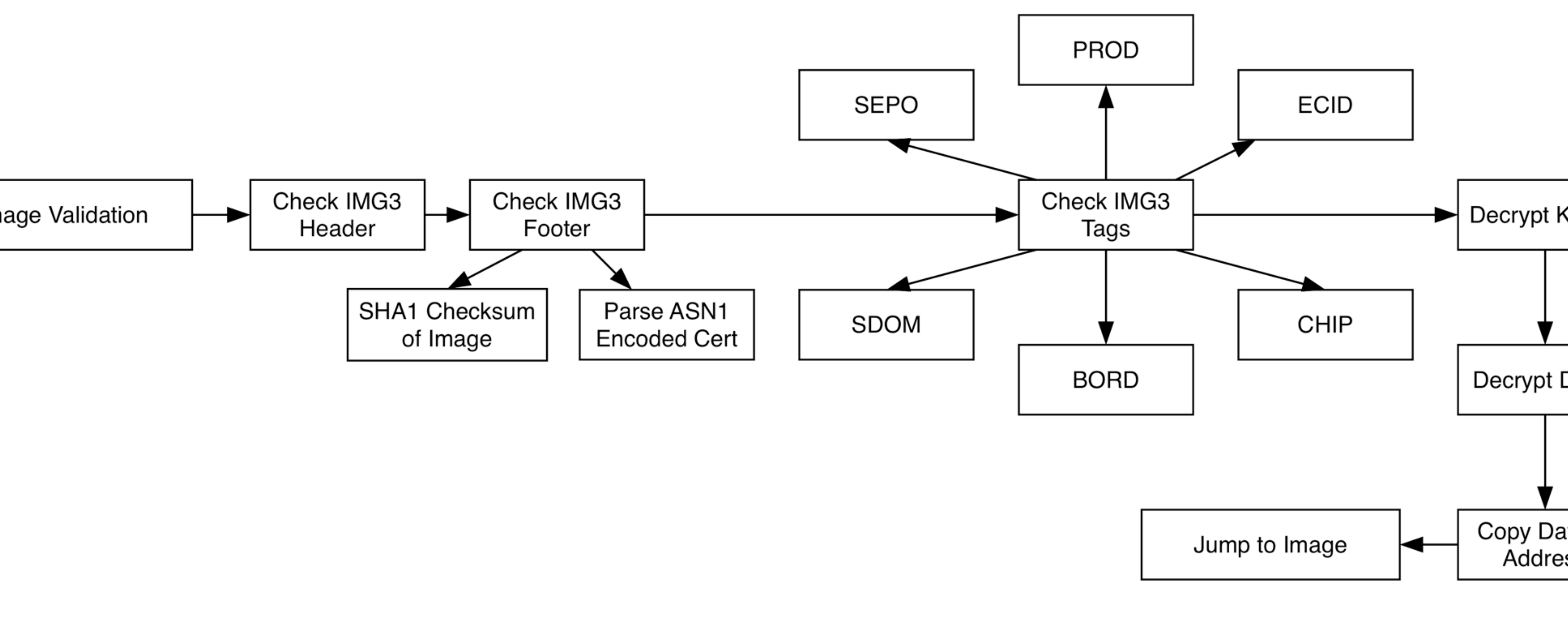
- Img3 header is sanity checked to ensure all sizes are in correct ranges.
- SHA1 taken of all data between the end of Img3 header, and the beginning of the SHSH tag.
- SHSH tag is decrypted by public certificate in CERT tag and verified against SHA1 hash of data.

Device Check

- Checks are performed to ensure image loaded is for correct chip, board, and version.
- ECID for device is checked to make sure firmware was personalized for this device only.
- These checks can all be bypassed if the device is a developer device.

Image Decryption

- KBAG tag in image is decrypted using the GID key in the AES module.
- Decrypted KBAG tag contains concatenated key and IV used to decrypt the DATA portion of image.
- If validation fails at any point, the entire image is cleared out and DFU mode is reentered.



age Validation
Check IMG3 Header
Check IMG3 Footer
SHA1 Checksum of Image
Parse ASN1 Encoded Cert
Check IMG3 Tags
PROD
SEPO
ECID
SDOM
BORD
CHIP
Decrypt K
Decrypt D
Copy Da
Addres
Jump to Image

Past Exploits in BootROM

- Pwnage2
- 24kpwn
- SteakS4uce
- SHAtter
- LimeRa1n

Pwnage and Pwnage2

- Pwnage based on fact Apple was not checking LLB's signature in BootROM allowing untethered jailbreak
- Pwnage2 discovered by Wizdaz allowed early code execution to apply first Pwnage.
- Vulnerability was in the certificate parser, but not much other information is known.

24kpwn

- Starting with iPod2g, Apple began checking signature of LLB and switched to new Img3 format killing Pwnage and Pwnage2.
- A new untethered BootROM exploit was needed.
- Chronic-Dev Team was formed and the search began.

The Discovery

- Shortly after I joined the search, @pod2g made an amazing discovery.
- The NOR Image loading routine was failing to check if the size of the image was larger than the size reserved for it.
- By flashing an LLB greater than 0x24000 bytes the end the image would begin overwriting the beginning of BSS segment.

The Analysis

- USB device descriptors and task structure had to be rewritten to prevent BootROM from becoming unresponsive or crashing.
- SHA1 MMIO addresses appeared to be a good target, but we were unsure how they were used.
- Finally @planetbeing came to the rescue!!!

The Exploitation

- Possible to achieve an arbitrary 4 byte write by overwriting SHA1 addresses.
- How do we know where exactly the return address is for us to overwrite though?
- BRUTE FORCE!!!!

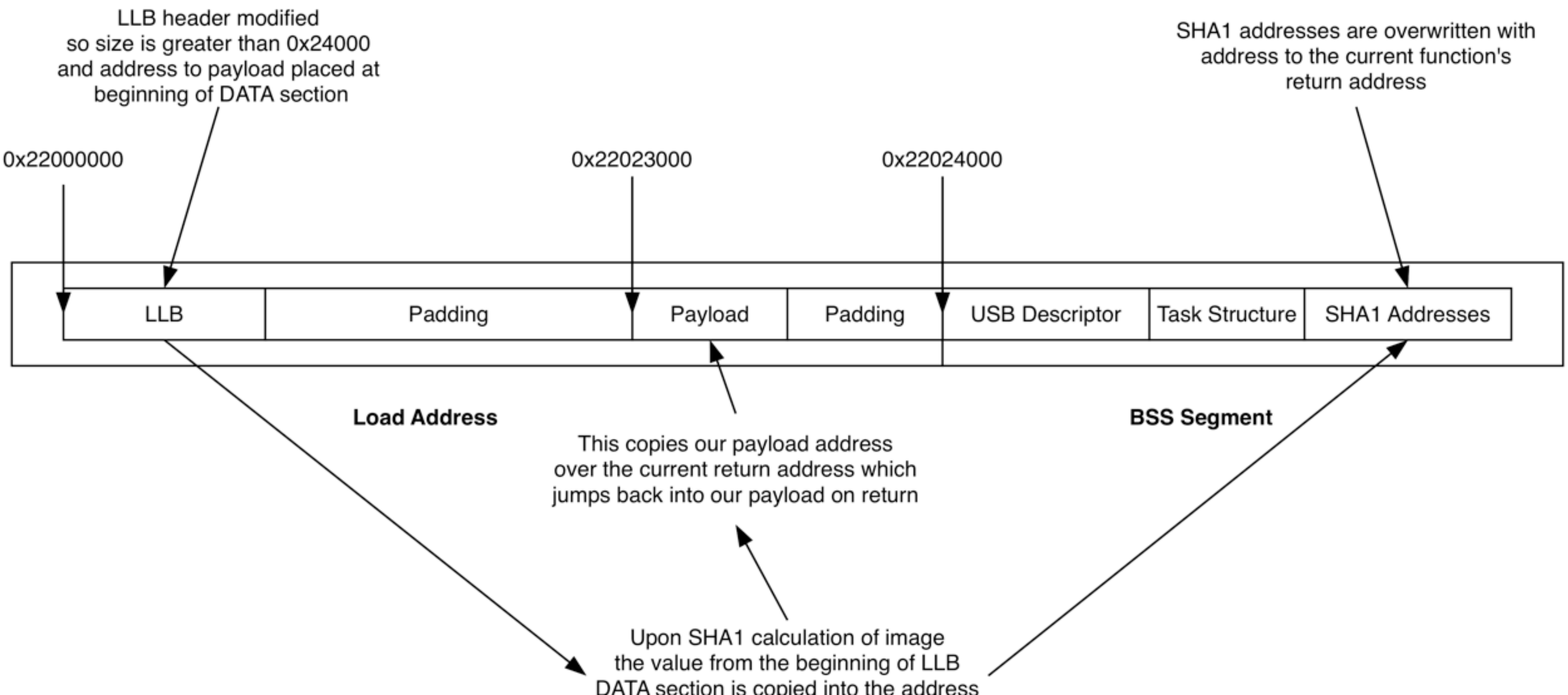
The Payload

- Payload was simple added into a known location in the exploit LLB.
- Fixed up memory that was altered to trigger the exploit.
- Finally jumps back into image loading routine past the signature checking to continue loading image unsigned.

The Big Picture

```
0x00: 33 67 6D 49 00 41 02 00 EC 40 02 00 8C 00 01 00
0x10: 62 6C 6C 69 41 54 41 44 0C 00 01 00 00 00 01 00
0x20: 01 30 02 22 35 98 E5 35 D8 56 21 DE 7A F2 6B 0A
0x30: AE 09 9D F8 26 C0 7A 1B 16 6F DC 2E FB 79 87 2A
```

```
0x240C0: C0 40 02 22 C0 40 02 22 C8 40 02 22 C8 40 02 22
0x240D0: 84 53 02 22 00 00 00 38 04 00 00 38 08 00 00 38
0x240E0: 0C 00 00 38 10 00 00 38 20 00 00 38 24 00 00 38
0x240F0: 28 00 00 38 2C 00 00 38 30 00 00 38 24 FE 02 22
```



SteakS4uce

- @comex enters scene and starts schooling us in userland exploitation.
- @pod2g decides to take another look into BootROM
- Comes to be excited saying he might of found one!

The Discovery

- A very simple USB fuzzer to try all possible USB packets.
- Sending A1, 1 packet seemed to be crashing all devices tested.
- @pod2g was sure it was a heap overflow.

The Analysis

- My analysis didn't show the same results.
- Eventually tracked it down on newer devices to an non-exploited double free.
- The fuzzing continued...

SHAtter

- After initial analysis the reversing of USB portion of BootROM we had a better understand of how things worked.
- I created a new fuzzer to attempt to see how USB packets were handled when the device was placed into different states.

The Discovery

- After sending 0xA1, 2 packet to the devices max size, failing validation and sending another an unexpected response was received.
- After this response the device crashed and rebooted.

The Response

```
ound iPhone/iPod in DFU/WTF mode
x84024000: 00 00 00 00 12 01 00 02 00 00 00 40 ac 05 27 12 ............@..'.
x84024010: 00 00 01 02 03 01 0a 06 00 02 00 00 00 40 01 00 .............@..
x84024020: 20 40 02 84 20 40 02 84 04 09 00 00 f4 49 02 84  @.. @.......I..
x84024030: 6b 73 61 74 00 00 00 00 00 00 00 00 0c 47 02 84 ksat.........G..
x84024040: 0c 47 02 84 03 00 00 00 03 00 00 00 f0 47 02 84 .G...........G..
x84024050: 2c 40 02 84 f0 47 02 84 7c 3f 03 84 1c 47 02 84 ,@...G..|?...G..
x84024060: 5c ae 00 00 00 00 00 84 00 40 02 00 70 3f 03 84 \........@..p?..
x84024070: fd 3e 00 00 00 00 00 00 00 00 00 00 02 82 3f 09 .>............?.
x84024080: 00 00 00 00 a0 86 01 00 00 00 00 00 5d 3d 00 00 ............]=..
x84024090: 30 40 02 84 94 40 02 84 94 40 02 84 00 00 00 00 0@...@...@......
x840240a0: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 ................
x840240b0: 62 6f 6f 74 73 74 72 61 70 00 00 00 00 00 00 00 bootstrap.......
x840240c0: 32 6b 73 74 c4 40 02 84 c4 40 02 84 f8 49 02 84 2kst.@...@...I..
x840240d0: f8 49 02 84 00 00 10 80 04 00 10 80 08 00 10 80 .I..............
x840240e0: 0c 00 10 80 10 00 10 80 20 00 10 80 24 00 10 80 ........ ...$...
x840240f0: 28 00 10 80 2c 00 10 80 30 00 10 80 40 00 10 80 (...,...0...@...
```

The Analysis

- For some reason BootROM was returning us the content from it's BSS, heap, and even stack segments!!
- After weeks of static analysis the reason was finally discovered.
- BootROM was failing to reset the index for the "Upload Counter" when reentering DFU mode.

Analysis Continues

- So we know why BootROM was returning us the data from BSS and heap segments, but why is it crashing?
- USB was returning this "Upload Counter" as the size of file being uploaded.
- When image validation failed, the image load routine was calling memset passing this size.

The Vulnerability

- Similar to 24kpwn exploit except we couldn't write arbitrary data, only zeros.
- To make things harder we could only write these zeros in 0x40 byte chunks.
- Non-exploitable you say? HA!!

First Attempt

- Change the return value!
- Could we overwrite the R4 register which was pushed onto stack before the call to memset?
- When memset returned R4 would be popped back off the stack and moved into R0 causing the image load function to return 0 (success)

First Fail

- Our payload, along with heap would been wiped out in the process.
- The 0x40 byte limit made targeting a specific register on stack unpractical.
- We needed to find a better way to control how much data was being zeroed

Second Attempt

- There was another memset in image loading routine.
- After SHA1 had been checked, the routine would memset over the data in SHSH tag.
- By altering the size of the SHSH tag we could then memset the exact number of bytes needed and leave our payload and image intact!

Second Fail

- Stack layout for this function was different than the previous memset.
- Heap had still be completely wiped out making recovery near impossible.
- Still unable to alter the return value to return success.

Third Attempt

- Since we could overwrite the exact number of bytes needed, why not attack heap?
- By overwriting the Least Significant Bytes we could alter heap address and point them back to a location we control!
- The address 0x840271C0 could become 0x84020000, right inside our load address.

Third Fail

- The code between the first SHSH memset, and the final memset at the end of image load routine was too short.
- Unable to find any usable pointers in heap to allow us to take control in this way.
- Sad and depressed we gave up.

Steaks4uce Revisted

- It appeared Apple had won that round.
- @pod2g turned his attention back to the heap overflow he discovered in iPod2g.
- In the process of exploiting it he stumbled upon another unexpected find.

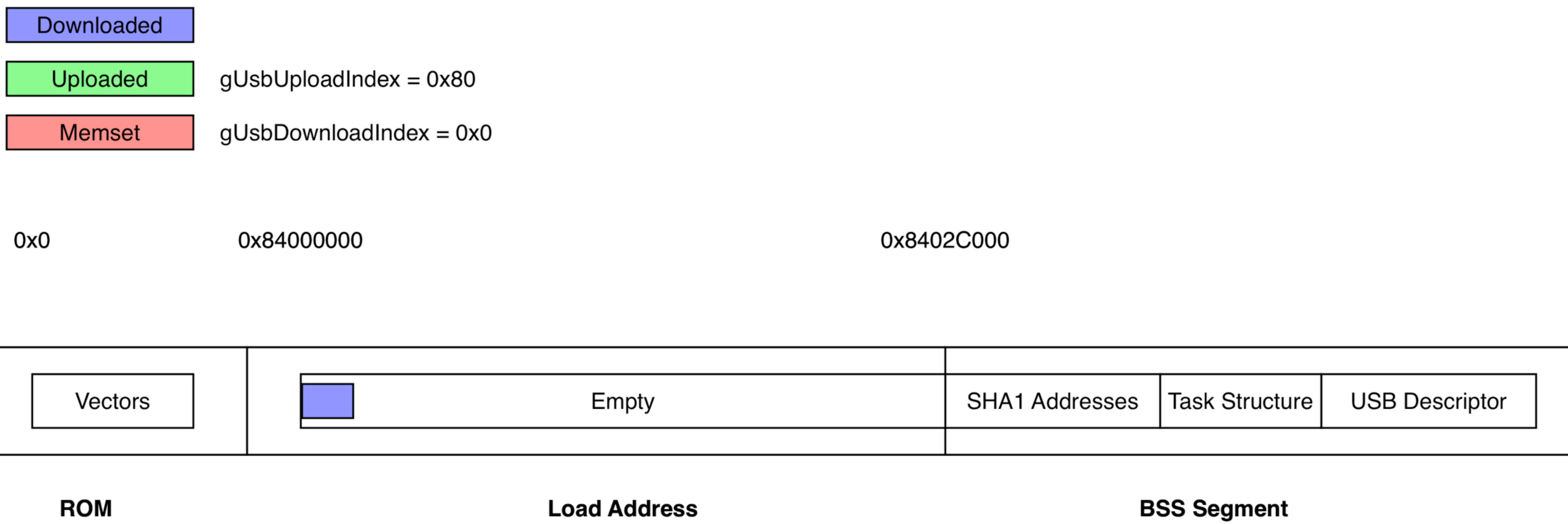
BootROM Exception Vectors

- While attempting to exploit SHAtter, we assumed overwriting a pointer to 0x0 would have no effect.
- 0x0 points to ROM, and there's no way to overwrite ROM right?
- Wrong!! The data containing the function pointers to exception vectors was actually writable!!!

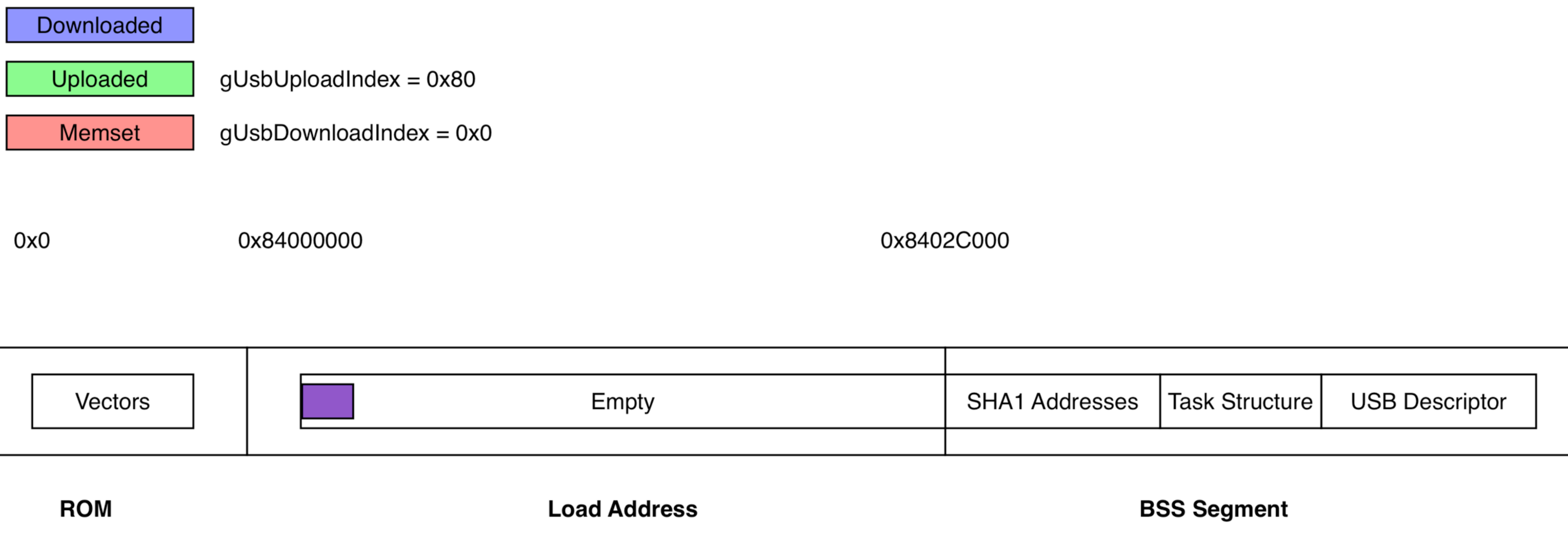
BootROM SHAttered

- The layout of BSS had changed in A4 BootROM.
- Instead of USB descriptors being the first structure in BSS, the SHA1 pointers were now the first values.
- By overwriting these to zero, we could then overwrite the exception vectors during the next SHA1 calculation!!

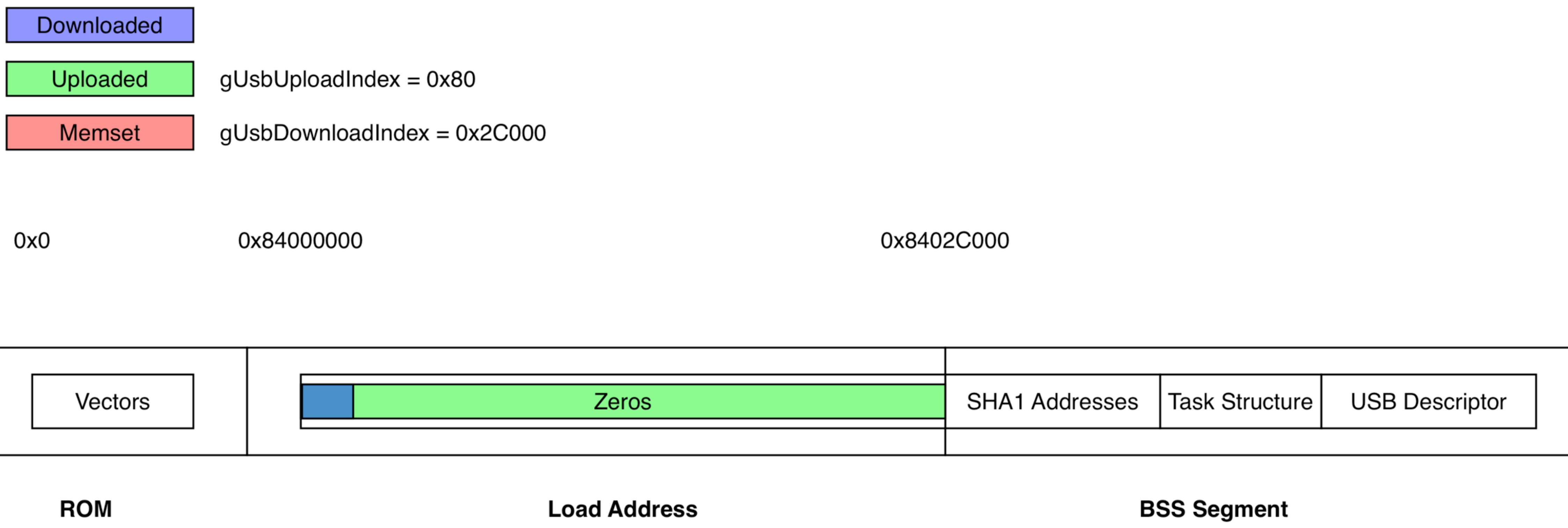
First task is to shift the upload index by 0x80 bytes



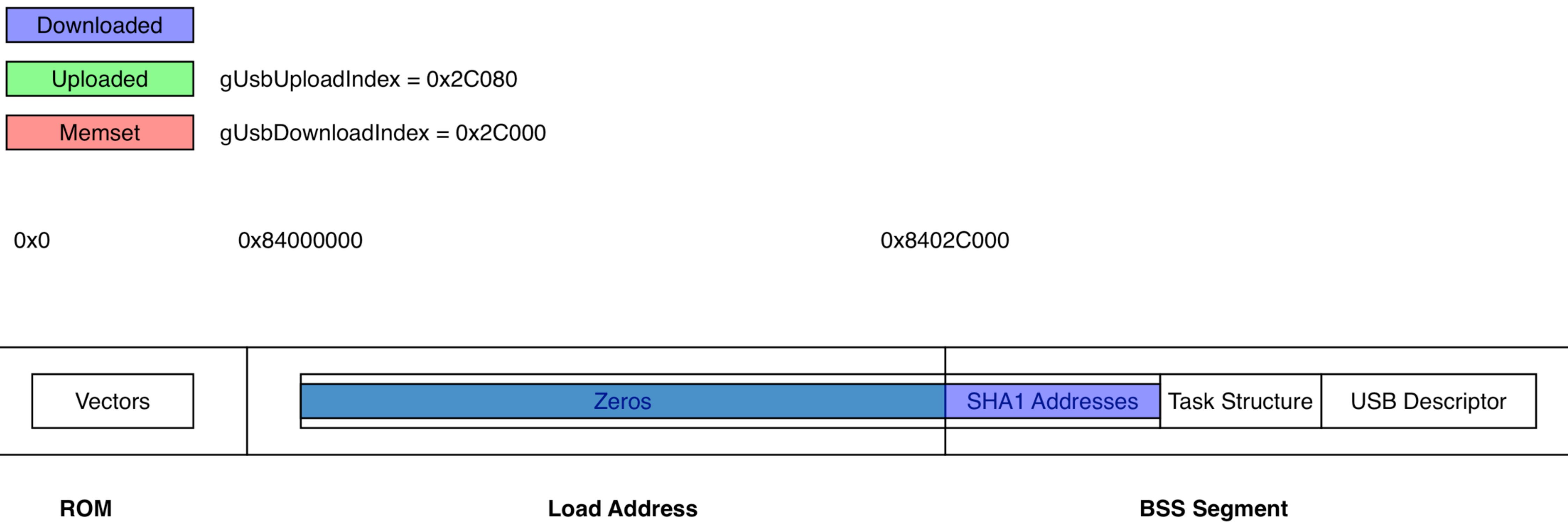
After a failed validation attempt 0x80 bytes is memset but the upload index isn't reset



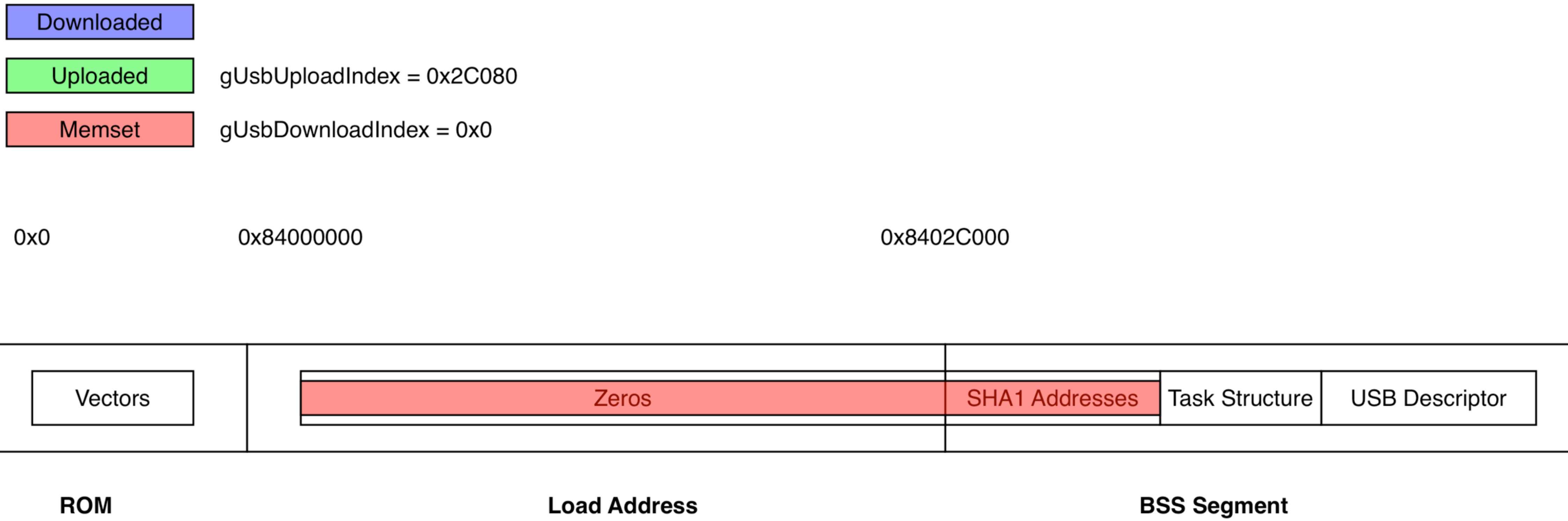
We now fill the buffer with zeros to ensure everything is set to a known value



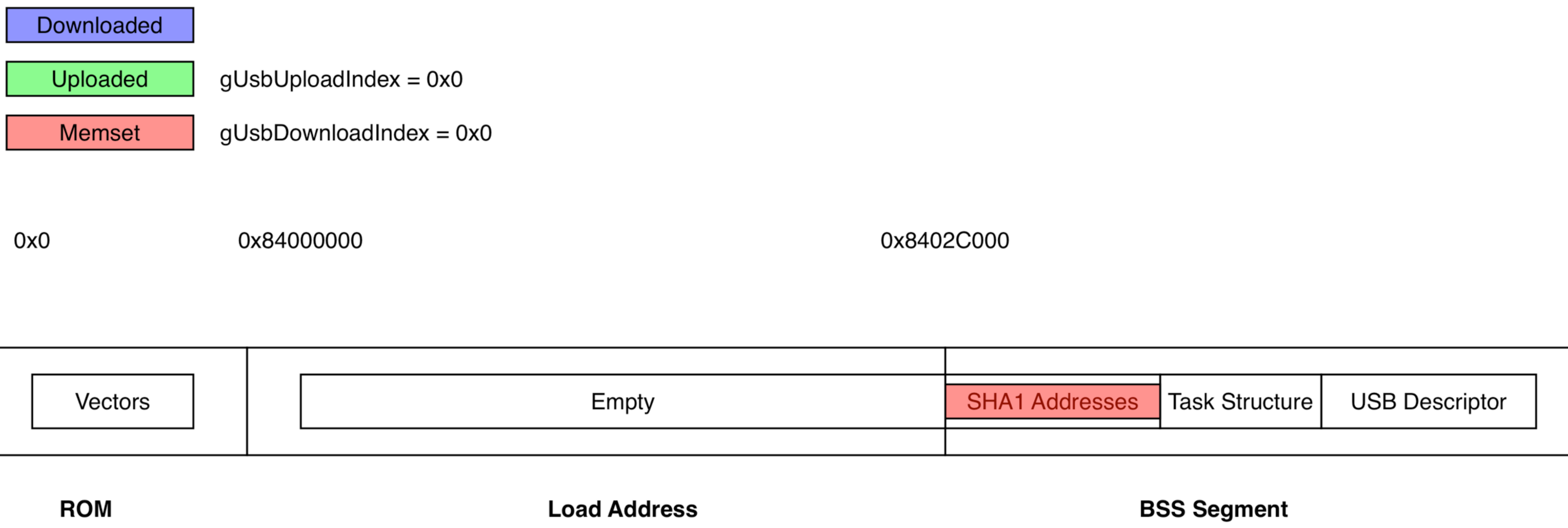
Next we download another 0x2C000 bytes from the device pushing the upload index 0x80 bytes past it's max size



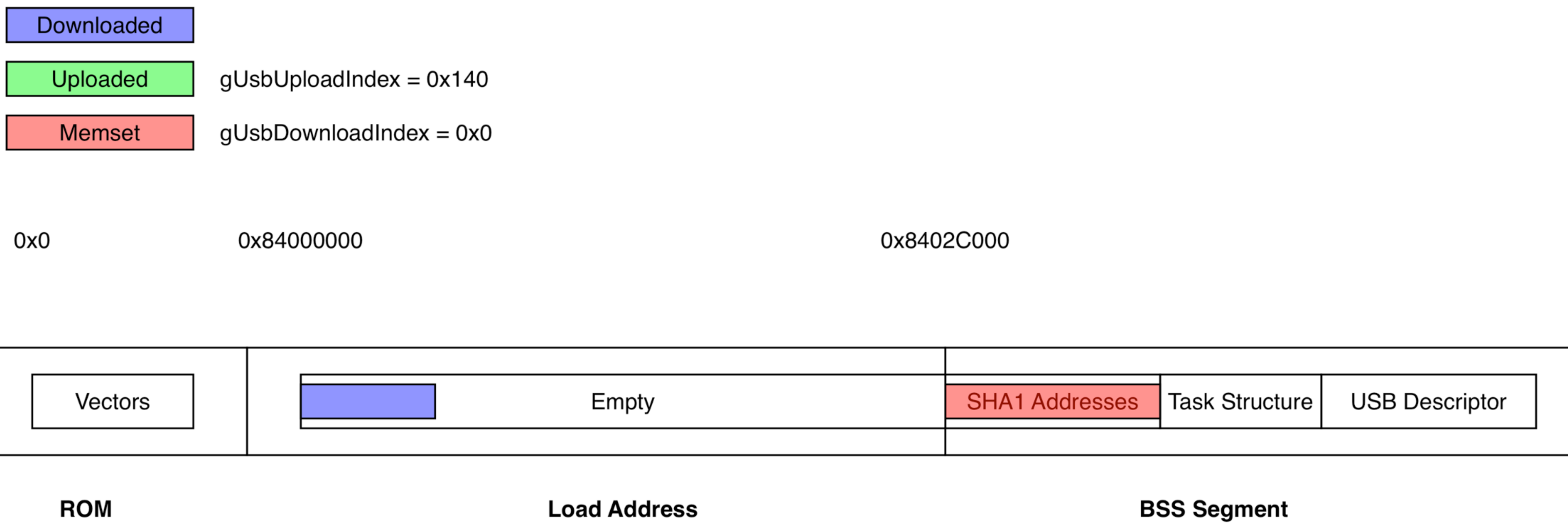
After another failed image validation attempt the SHA1 registers are overwritten with zeros



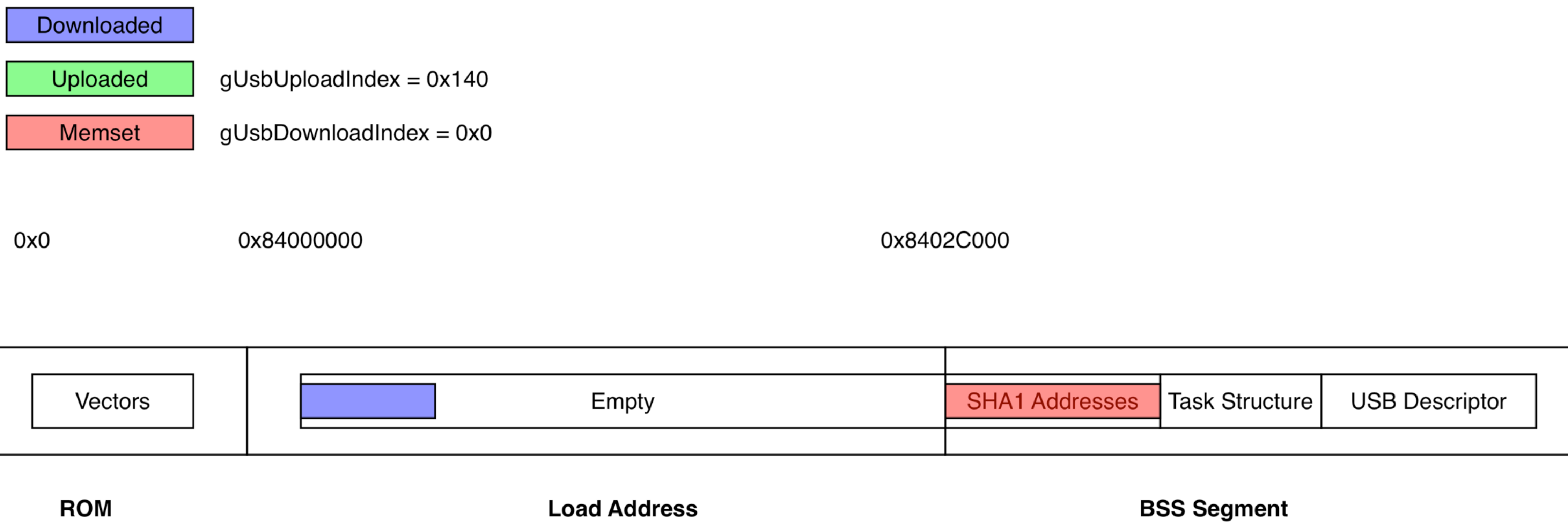
The counters are reset to prepare for the second pass



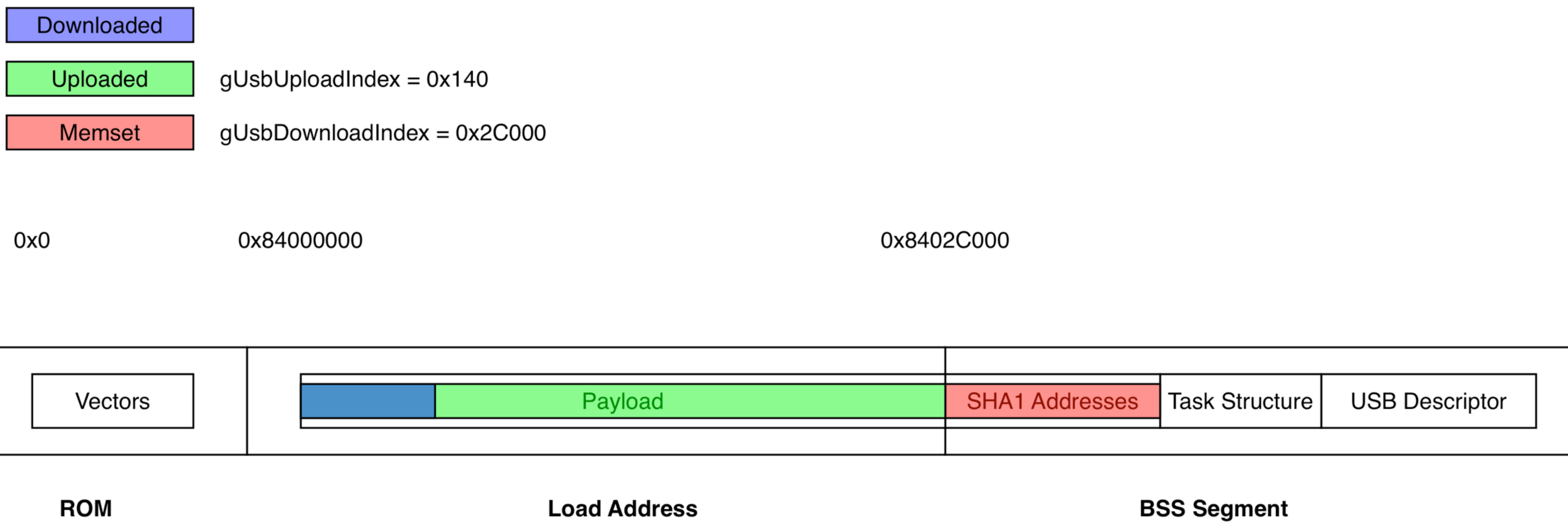
This time we shift the upload index by 0x140 bytes



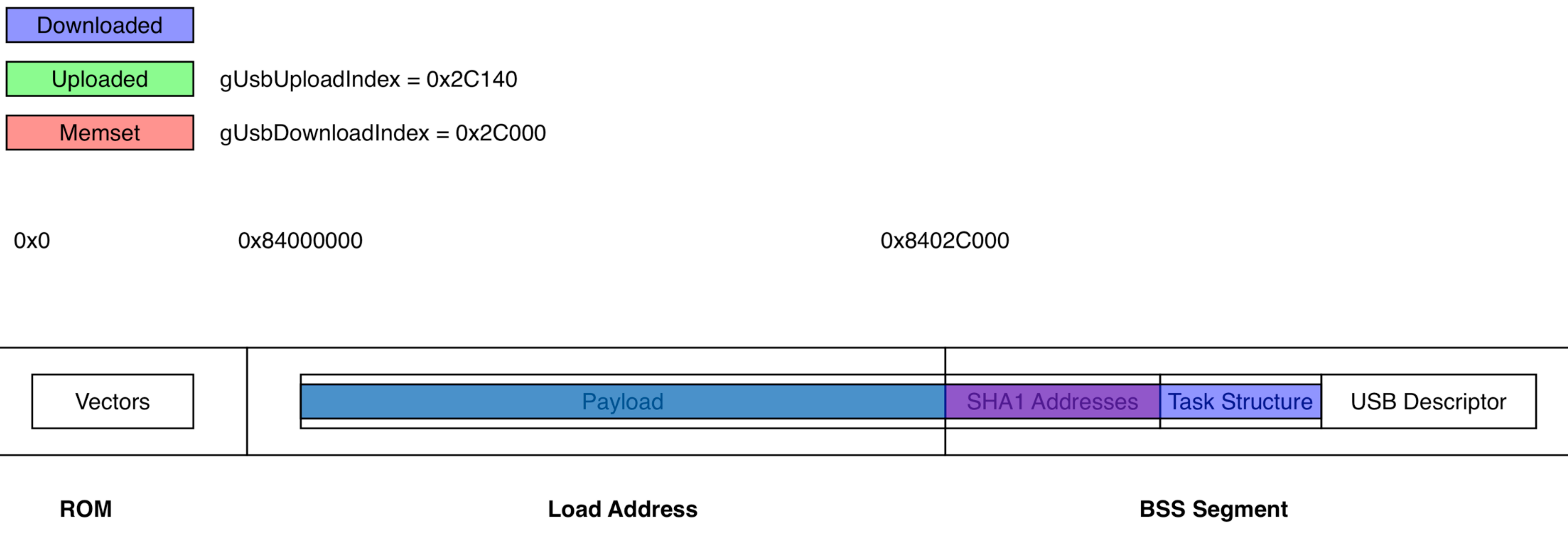
After another failed image validation attempt the upload index remains at 0x140



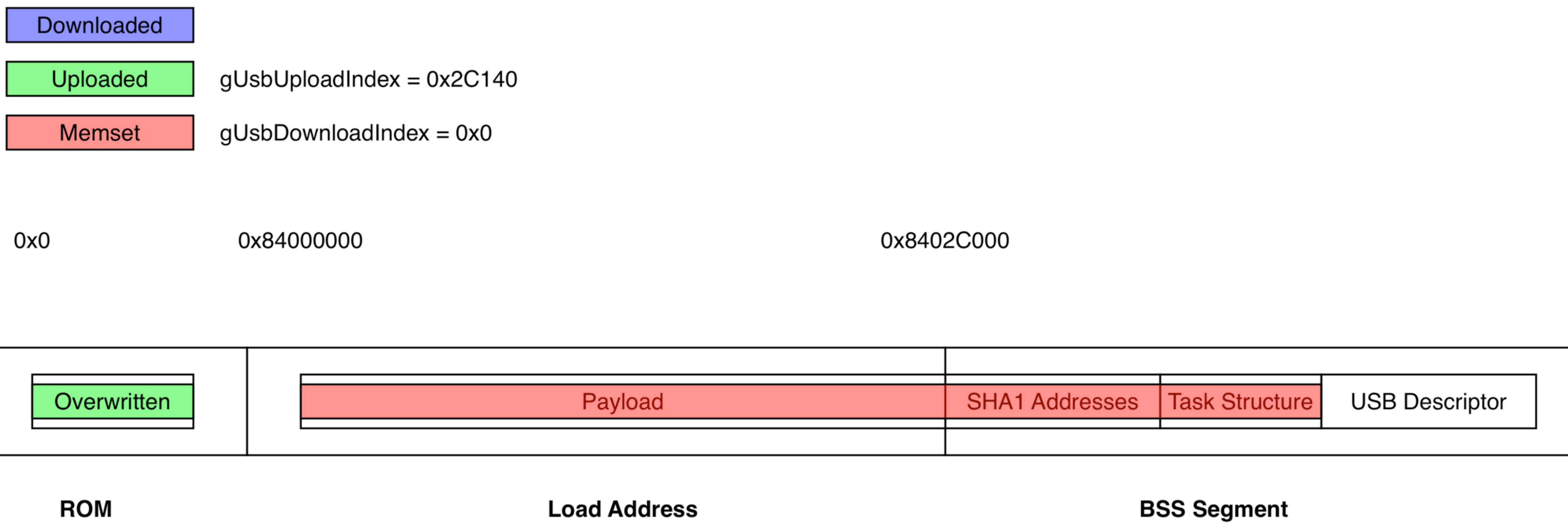
Finally we upload our payload containing the fake exception vectors pointing to our payload



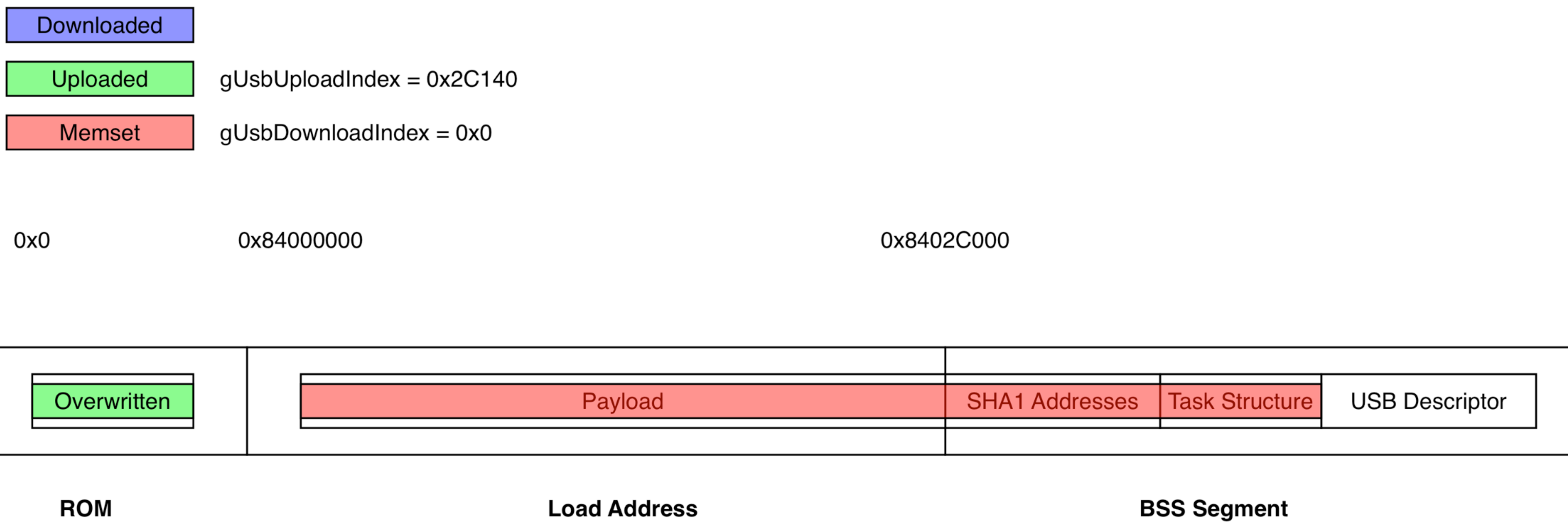
Last we download another 0x2C000 bytes to push the size up to 0x2C140



This time when image validation occurs the exception vectors are overwritten with the data in our payload



When the memset occurs at the end a panic occurs and our new exception handler is called to jump to our payload



The Payload

- First it accepts an image to be sent over USB.
- Next it decrypts that image manually.
- Finally it patches the image to remove signature checks and change the address of "go" command.
- Finally we jump into our unsigned image.

The Tragedy

- Spent over a month fixing GreenPois0n to use SHAtter exploit.
- Announced we'd finally be releasing GreenPois0n on 10/10/10 at 10:10:10
- Three days before release, @geohot pops up and releases LimeRa1n exploit.

LimeRa1n

- After announcing greenpois0n release date, @geohot thought we had discovered the same exploit as him.
- Although @geohot had also discovered SHAtter he didn't think it was exploitable.
- LimeRa1n was superior since he worked on all devices and SHAtter only worked on A4.

The Discovery

- Not much is known how he discovered it.
- He probably was just fuzzing USB packets like we were.
- USB timeouts were broken in libusb on OSX so we would of never found this vulnerability.

The Analysis

- LimeRa1n appears to be a race condition heap buffer overflow in USB stack.
- After release I asked @geohot to explain why it worked.
- He said he had no clue, but I will speculate on my theory in the next part.

The Exploit

- By sending a packet with a short timeout (10ms) heap corruption allowed an arbitrary 4 byte overwrite.
- SHAtter was used to locate the return address to overwrite.
- Spray the heap with fake chunks and wait for something to be freed.

The Payload

- Biggest pain was creating a work-around for libusb's broken timeouts.
- After replacing SHAtter with LimeRa1n in greenpois0n we just used the same payload.
- We hoped we could keep SHAtter private for the next devices, but found it posted on pastebin the next day.

BootROM Exploitation Methods

- Stack Buffer Overflows
- Heap Buffer Overflows
- Segment Buffer Overflows
- Race Conditions
- Recursive Stack Overflows

Stack Buffer Overflows

- Very easy to exploit if discovered.
- Stack is executable and deterministic.
- Payload could also be place and executed in load address or heap.

Heap Buffer Overflows

- Not much more difficult to exploit.
- Heap is executable.
- Few allocations also make heap very predictable.
- Only challenge is finding return address on stack.

BootROM Heap Simulator

- During SHAtter I reverse engineered the allocation functions and created a simulator.
- Very few allocations in heap make it very predictable.
- Let’s you visualize and debug heap layouts in BootROM to create heap overflows.

Segment Overflows

- The type of bug we've encountered most often.
- With arbitrary control very easy to exploit.
- With limited control of data, exploitability depends on what's contained in next segment.
- SHA1 MMIO address always a good target.

Race Conditions

- Only 2 tasks running in BootROM idle_task and usb_task.
- Hardware interrupts can also be seen as tasks.
- Software can't predict when hardware will send an interrupt (unless you're sitting at a WFI instruction).

The Theory

- USB packet sent to the device.
- IRQ exception is thrown and USB interrupt handler launched.
- Packet is queued and control returned to main task.
- Main task begins to handle this packet.

The Panic

- During processing of USB packet, another USB packet is sent which clears the queue.
- Control is returned back to main task which unknowingly continues trying to handle packet which was deleted.
- This is most likely the reason behind the LimeRa1n exploit.

Recursive Stack Overflows

- Not sure of any recursive functions in BootROM, but there might be.
- Main stack is fairly large and might be difficult to pass.
- Exception stack is much smaller and boarders the end of heap.

Summary

- Limited attack surface, but most crashes found were exploitable.
- Difficult part is lack of debugging and months of static analysis.
- Hopefully more people will be interested in helping find new BootROM exploits.

Questions?