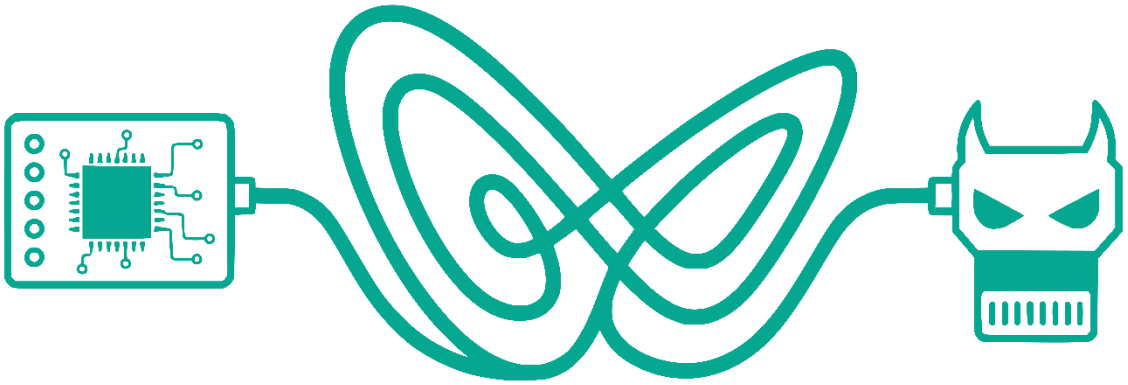
Introducing usbliter8

An A12/A13 SecureROM exploit



TLDR - here is the PoC

This write-up details a novel iPhone BootROM vulnerability discovered and exploited by our team. It covers the underlying bug, the associated exploitation techniques, and the post-exploitation steps required to achieve application processor's boot-chain compromise. The exploit leverages both a hardware bug in the USB controller and a specific configuration flaw present in the device firmware.

Currently supported SoCs include Apple A12, S4/S5, and A13. While technical support for A12X/Z is possible, it is not currently implemented. We limited our implementation to these devices, as demonstrating successful exploitation across this range was sufficient to thoroughly validate both the vulnerability and the exploitation strategy.

By publishing this research and the accompanying proof of concept, we aim to document the real-world impact of this class of hardware vulnerabilities, contribute to the broader understanding of modern BootROM security, and demonstrate that even recent SecureROM generations remain susceptible to subtle hardware flaws.

As these vulnerabilities reside in immutable code, affected users should be aware that migrating to newer hardware remains the most effective mitigation.

USB Setup Packets Anatomy

In the context of the USB specification, every control transfer must initiate with a Setup transaction. This is the mechanism the host uses to issue any kind of request to the attached device.

A proper Setup transaction consists of two packets sent by the host:

(1) TOKEN PACKET host → device

SYNC	PID	ADDR	ENDP	CRC5	EOP
8 bit	8 bit	7 bit	4 bit	5 bit	
	(SETUP)	device	endpoint		

(2) DATA PACKET host → device

SYNC	PID	DATA (8 bytes) = USB Device Request	CRC16	EOP
8 bit	8 bit	16 bit		
	(DATA0)	This is what the USB driver receives		

According to the specification, the data payload of a Setup transaction must be exactly 8 bytes and adhere to a strict format. The device request structure contained within this payload is passed verbatim to the software driver for handling.

To maintain clarity in our analysis, we will refer to this specific payload as the "Setup packet."

DWC2 Direct Memory Access

The USB controller used by Apple in their SoCs is the DWC2 by Synopsys. Detailed information on how this controller works can be inferred by analyzing other existing driver implementations, such as those found in the Linux kernel.

What we care about here is how the controller writes data to main memory. The AP configures DMA by allocating a memory region and writing its physical address to a specific register (DOEPDMA) in the controller's MMIO region.

The controller uses this buffer to store data received in SETUP and OUT packets, which is then processed by the device.

We did observe that when the USB controller writes data chunks, it directly increments the address stored in the DOEPDMA register. This is a crucial detail since it might mean that the register value acts as a direct source of truth, defining the physical address that will be used for the next DMA transfer, instead of just being a configuration facility.

The Bug

The DesignWare USB controller stores up to three consecutive Setup packets in memory.

Upon receiving a fourth Setup transaction, the DMA base address gets reset to its starting position before writing, akin to a ring buffer mechanism.

After writing each received packet, the controller increments DOEPDMA by the size of data written. The reset operation is implemented by decrementing DOEPDMA by 24.

The core issue arises because the controller also accepts smaller packets (though always stores in 4-byte chunks).

Since the pointer increment does not match the fixed decrement amount, we end up with a buffer underflow primitive in 12-byte steps.

We believe this is an inherent bug within the USB controller itself. While potentially affecting many devices, the vulnerability works under specific circumstances only.

As of today, we have confirmed that the A12 and A13 SecureROMs are vulnerable, whereas A11 is not. The difference is that the A11 USB driver manually resets the DMA address to its initial value after receiving each packet.

On A12 and A13, USB DART is configured in bypass mode, allowing us to overwrite SRAM data freely. In contrast, A14 and later generations appear to configure the DART correctly in SecureROM, making the vulnerability unexploitable.

PC control on A12

Achieving PC control on A12 is straightforward because the USB controller's DMA buffer is allocated on the heap shortly after the USB task's stack.

The simplest approach is to overwrite a saved LR on the stack and obtain direct PC control when the scheduler performs a context switch back to the USB task.

PC control on A13

Things aren't as easy on A13 SecureROM due to the introduction of PAC.

From what we observed, PAC appears to be applied only to stack-stored LRs, which however is enough to prevent us from directly targeting the USB task's saved LR as we did on A12.

Several mitigations had to be bypassed along the way. These include heap metadata checksums, which are verified during heap operations, and LR signing during context switches, which occur whenever the USB task is woken up to process USB packets.

After some iterations, we came up with the following multi-step technique to achieve PC control.

The first step is to overwrite some DART-related data located in the heap immediately before the USB controller's DMA buffer. This gives us very limited write primitives that can be triggered once when exiting the DFU loop.

We leverage some of the cleanup routines, which use controlled data, to zero out the global pointer to the DART allocation. This step is necessary to prevent the corrupted allocation from being freed, which would otherwise trigger a panic when the heap checksum is checked.

```
void dart_stop(unsigned int dart_id)
{
 // [1] we can fully overwrite the heap memory for this object
 dart = darts[dart_id];
 mmio_base = dart->info->mmio_base;
 v4 = 16 * dart->ctx->field_11 + 0x200;
 for ( int i = 0; i < 16; i += 4 ) {
  // [2] this gives us a 16-byte zero write primitive
  *(_DWORD *)(mmio_base + v4 + i) = 0;
 }
```

```
  dart_flush_maybe(mmio_base);
}

void dart_free(__int64 dart_id)
{
  // [...]
  dart_stop(dart_id);
  enter_critical_section();
  ref_count = info->ref_count - 1;
  info->ref_count = ref_count;
  if ( !ref_count )
    irq_mask(info->int_irq_num);
  exit_critical_section();
  // [3] we need to use the zero write primitive for these second deref to return 0 and make the free a no-op
  dart = darts[dart_id];
  free(dart);
  darts[dart_id] = 0;
}
```

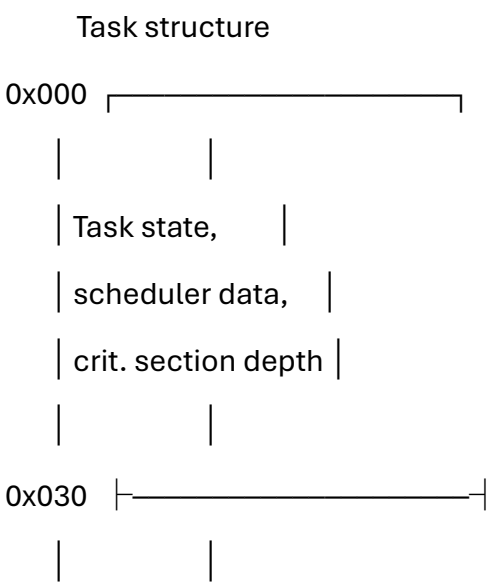
Next, as part of the same cleanup path, we use a 0xf write primitive to overwrite a global panic counter. In this way, the next panic will cause the CPU to enter an infinite loop rather than triggering a reboot.

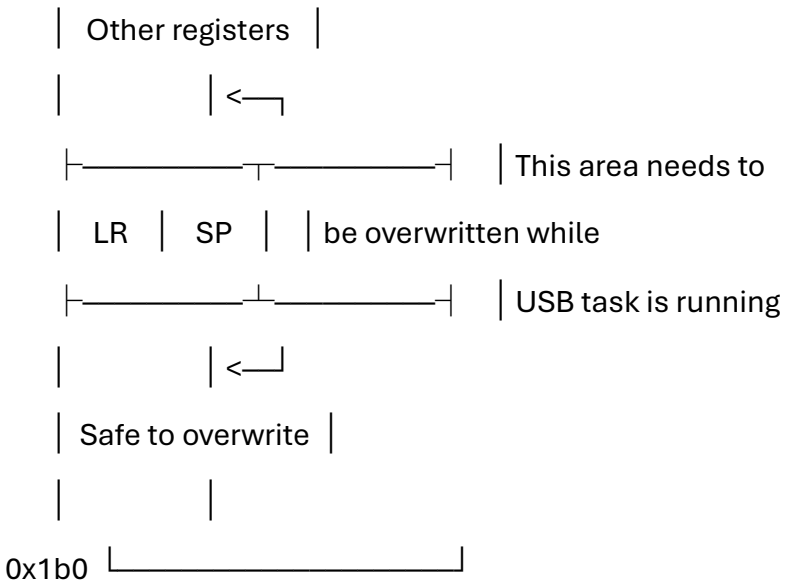
```
void dart_flush_maybe(__int64 mmio_base)
{
  __dmb();
  // [4] these gives us a 0xF write primitive targeting the panic depth counter
  *(_DWORD *)(mmio_base + 52) = 0xF;
  *(_DWORD *)(mmio_base + 32) = 0;
  ticks = get_ticks();
```

```
 while ( 1 ) {
  v3 = get_ticks();
  if ( (*(_DWORD *)(mmio_base + 32) & 4) == 0 )
    break;
  if ( v3 - ticks >= 1000000 )
    panic();
 }
}

void __noreturn panic()
{
 // [5] after overwritting this global we would enter an infinite loop on panic
 if ( ++panic_cnt >= 3 )
  spin();
 // [...]
}
```

The next step is to avoid breaking the USB task's context, in particular the LR and SP registers, which are saved to and restored from the task structure during context switches. To achieve this, we time DMA writes to happen when the the task is awake, so that when it yields, the correct register values overwrite the ones we corrupted.





After this, we can target a field within the task structure itself that tracks the task's critical-section depth. This allows us to trigger a panic with IRQs enabled, causing execution to enter the infinite loop established in the first step while still allowing ISRs to run. Additionally, the USB controller remains in a state that allows us to continue writing data to memory.

```
void enter_critical_section()
{
  current_task = current_task();
  critical_section_depth = current_task->critical_section_depth;
  if ( critical_section_depth < 0 || critical_section_depth >= 10000 )
    panic();
  // [1] on each call increments task's critical_section_depth
  current_task->critical_section_depth = critical_section_depth + 1;
  if ( !critical_section_depth ) {
    irq_disable();
  }
}

void exit_critical_section()
{
```

```
 current_task = current_task();
 // [2] should be equal to the amount of entries to `enter_critical_section`
 // but we can overwrite it with a smaller count
 critical_section_depth = current_task->critical_section_depth;
 // [3] on first entry it will enable interrupts and on second entry it will panic
 if ( critical_section_depth <= 0 )
   panic();
 critical_section_depth = critical_section_depth - 1;
 current_task->critical_section_depth = v2;
 if ( !critical_section_depth )
   irq_enable();
}
```

After all of this setup, we can freely overwrite memory until we reach the global variable containing the USB IRQ handler itself. By overwriting it with an arbitrary value, we gain PC control.

```
// [1] an array of `irq_handler_ctx` structs lives in the BSS section which we can reach with our bug
00000000 struct irq_handler_ctx // sizeof=0x18
00000000 {
00000000     void (*handler)(void *arg);
00000008     __int64 *arg;
00000010     _BYTE unk;
00000011     _BYTE shall_mask;
00000012     // padding byte
00000013     // padding byte
00000014     // padding byte
00000015     // padding byte
00000016     // padding byte
00000017     // padding byte
```

```
00000018 };


void handle_irq()
{
  irq_num = MEMORY[0x23B102004];
  if ( MEMORY[0x23B102004] ) {
    while ( irq_num ) {
      if ( (irq_num & 0x70000) != 0x10000 )
        panic();
      irq_num__ = irq_num & 0x1FF;
      // [2] after the Setup we can freely overwrite handler for USB interrupt with a
controlled value
      handler = irq_list[irq_num__].handler;
      if ( handler )
        // [3] fully controlled handler gets called with fully controlled arg
        handler(irq_list[irq_num__].arg);
      _clr_mask_irq(irq_num__);
      irq_num = MEMORY[0x23B102004];
    }
  } else {
    ++stale;
  }
}
```

Post-Exploitation

Starting with A12, SecureROM mostly runs in EL0 mode. This means we cannot access protected memory regions (namely the MMU tables and the ROM's associated metadata) or special CPU registers (such as SCTLR, TTBR, and others).

SecureROM can switch execution to the privileged EL1 mode by executing the SVC 0 instruction. This is not a syscall-like interface: the handler simply returns to the code

immediately after the SVC instruction, but with execution continuing in EL1 mode. Shortly afterward, the ROM executes an ERET to continue in EL0.

There are very few places in the ROM where this transition occurs. We chose one in the function responsible for transferring execution to the next-stage iBoot:

A12 SecureROM

...

1000088B0: SVC 0 # switching to EL1, we want to land here

1000088B4: LDR W8, [X22]

1000088B8: CBNZ W8, loc_10000891C

1000088BC: MOV W8, #1

1000088C0: STR W8, [X22]

1000088C4: STRB W8, [X23]

1000088C8: ADRP X9, #0x19C014033@PAGE

1000088CC: LDRB W10, [X9,#0x19C014033@PAGEOFF]

1000088D0: CBNZ W10, loc_10000891C

1000088D4: STRB W8, [X9,#0x19C014033@PAGEOFF]

1000088D8: LDRB W8, [X24,#0x19C014032@PAGEOFF] # unintended compiler behavior, described later

1000088DC: CBZ W8, loc_10000891C

1000088E0: BL sub_100000430 # clearing some SCTLR bits

1000088E4: BL sub_100006798 # clearing something in scratch regs?

1000088E8: BL sub_10000739C # returns boot trampoline address

1000088EC: MOV X8, X0

1000088F0: CBZ X8, loc_100008904

1000088F4: MOV X0, X20

1000088F8: MOV X1, X19

1000088FC: BLR X8 # jump to the boot trampoline!

...

A12 & S4/S5

These SoCs do not use PAC in SecureROM. We gain code execution by corrupting LR on the stack. Time for a good ole ROP!

The chain is very small - just a few frames:

1. Perform a 32-bit write to the USB MMIO to change the DMA destination address
2. Sleep a bit (~400ms)
3. After sleeping, jump to 0x1000088B0 (the assembly above)

The DMA destination is set to a location within the boot trampoline. This memory is obviously not writable from EL0 (or even EL1, as it is intended to be executable-only), but since it's DMA...

The boot trampoline is also always stored at the same address.

The delay gives the exploit enough time to write our shellcode into the boot trampoline.

After the delay, we finally perform the jump. The firmware actually contains checks to prevent jumping somewhere in the middle of the function. It maintains a series of state flags that are updated after each step. If the flag corresponding to a previous stage is not set, the firmware triggers a panic.

Interestingly, the compiler generated the following instructions for these checks:

...

1000088C8: ADRP X9, #0x19C014033@PAGE # get page addr for flags' area into X9

1000088CC: LDRB W10, [X9,#0x19C014033@PAGEOFF] # fetching something via X9

1000088D0: CBNZ W10, loc_10000891C # this check doesn't fail, luckily

1000088D4: STRB W8, [X9,#0x19C014033@PAGEOFF] # storing some other flag, also via X9

1000088D8: LDRB W8, [X24,#0x19C014032@PAGEOFF] # doing a dangerous check... via X24!

1000088DC: CBZ W8, loc_10000891C

...

X24 is a callee-saved register, which means its value is stored on the stack. That very stack we fully control, yes.

At this point we get to execute our own code with EL1 privileges. All we have to do now is to clean up, apply our changes and return to DFU like nothing happened.

Long story short:

1. The original boot trampoline needs to be copied back where it belongs
2. All the heap allocations that we corrupted need to be restored; luckily, there are only a few, and they contain static data
3. We want to inject our own USB control request handler; we place it in unused space within the boot trampoline area and overwrite the corresponding callback pointer in BSS
4. Inject the PWND string into the USB serial number string
5. Carefully repair the usb_task() stack frame and return execution to it

Done! We now have DFU mode running with our own request handler.

Of course we skipped many important implementation details, but the full exploit is available for reference in our PoC repository.

The handler

Very simple code. All standard USB DFU requests are forwarded straight to the original handler. However, we add a few custom requests:

1. Demotion: lowers the SoC's production mode (temporarily, of course, until the next reboot)
2. Booting iBoot: allows raw iBoot images to be booted without any signature checks

There was a funny issue with booting iBoot at first. If we call the jump function directly from our handler, it does not work. The jump function quiesces various hardware components, including USB, which causes the USB task to terminate.

Yes, that very USB task we are executing from...

So, instead of calling the jump function directly, we overwrite the main task's return address on the stack and then close the DFU session. The USB task shuts down, but the main task is woken up and resumes execution. Here is the place where we reroute it to:

```
# A12 SecureROM

...

100001C8C:      BL          sub_100008F60       # something we cannot really skip

100001C90:      BL          sub_100008FE4       # same here

100001C94:      TBNZ        W28, #2, loc_100001C9C  # check if it's local boot
```

```
100001C98:      BL          sub_100006850       # set scratch reg to signify remote booting for next stage,
                                                # we do it inside of the handler code unconditionally
100001C9C:      MOV         X1, #0x19C030000    # load address, our raw iBoot will end up there
100001CA4:      MOV         W0, #0
100001CA8:      MOV         X2, #0
100001CAC:      BL          jump_to_next_stage
...
```

A13

A13 is a whole different beast due to the introduction of PAC. However, the post-exploitation strategy remains pretty much the same.

We cannot just ROP here, but luckily the boot trampoline area sits between BSS and heap, allowing us to overwrite it with controlled data midway through the exploit.

The jump function suffers from a problem similar to A12: the address used for the *steps validation* is saved on the stack (X22). Unfortunately, we cannot easily control it in this case.

Instead, we found a very nice gadget:

```
# A13 SecureROM
...
10000A5E8:      MOV         X22, X0             # land here, immediately get controlled X22
10000A5EC:      ADRP        X10, #__stack_cookie@PAGE
10000A5F0:      NOP
10000A5F4:      LDR         X10, [X10,#__stack_cookie@PAGEOFF]
10000A5F8:      STUR        X10, [X29,#var_38]
10000A5FC:      LDR         X10, [X22]
10000A600:      ADD         X10, X10, #0xF
10000A604:      AND         X10, X10, #0xFFFFFFFFFFFFFFF0
10000A608:      MOV         X11, SP
```

```
10000A60C:      SUB         X23, X11, X10
10000A610:      MOV         SP, X23                 # all this logic doesn't break anything, luckily
10000A614:      LDR         X10, [X22,#0x10]        # oh look, function pointer from a controlled place!
10000A618:      MOV         X1, X23
10000A61C:      MOV         X2, X9
10000A620:      MOV         X3, X8
10000A624:      BLRAAZ      X10                     # jumping to the jump func, auth is fake
...
```

It is interesting that the firmware makes extensive use of authenticated branch instructions, yet only the IB key is actually enabled. Well, good for us!

At this point, we execute our own code with full privileges.

Cleaning up on A12/S4/S5 was already challenging. On A13 the situation is even worse since we ended up destroying almost everything...

ROM Restart

The way we do it here is by restarting SecureROM and letting it reinitialize everything for us. The only problem is that we also want that our changes survive the restart.

To achieve this, we copy the ROM to the very end of SRAM, so we can patch it any way we want.

Unfortunately, the ROM expects to execute from its original address at 0x100000000, not from SRAM. It often performs PC-relative accesses into SRAM (0x19C000000), which would then go nowhere if ROM runs from an unexpected location. In order to circumvent this, we configure the MMU in such a way that the SRAM physical addresses we copied the code to will map to the original ROM virtual addresses.

The MMU will be enabled from the very beginning of our in-SRAM ROM. At some point, it will create its own translation tables and switch to them. At that exact moment, everything would fall apart. However, we can hook ROM PTE generation and still route it the way we want.

There is quite a bit of *dance* in the code that performs the ROM restart. Check the source code for the full reference.

Speaking of patches, here is a list:

1. Lower load area size. Our remapped ROM technically belongs to the load area (where raw data loaded from NAND or USB goes), so the ROM will try to clear it. This would be a disaster. Therefore, we patch the ROM to not consider the last few hundred kilobytes of SRAM, so it won't be touched.
2. Force DFU. We obviously want to return to DFU and not boot whatever resides in NAND.
3. Hook USB serial number creation to add the PWND part.
4. Inject our USB request handler.

That's it. SRAM is pretty big nowadays, and so is the load area. Stealing a few hundred kilobytes from it shouldn't cause any problems.

Conclusion

The usbliter8 exploit demonstrates that even on more recent SecureROM generations, including those protected by Pointer Authentication, subtle hardware bugs can still be leveraged to achieve full code execution and break the chain of trust.

The security of the BootROM is critical: vulnerabilities at this level can compromise the integrity of the entire device. However Apple's SEP (Secure Enclave Processor) adds an additional security boundary between attacker and user data.
Although usbliter8 doesn't affect SEP itself, it opens up wider attack vectors to compromise the Secure Enclave. By releasing this exploit publicly, we hope to highlight the real-world impact of these hardware flaws and contribute to a broader understanding of modern SecureROM security.

While newer generations have addressed the underlying issue, affected A12 and A13 devices will carry it for the remainder of their lifetime. For those who have followed the history of iPhone exploitation and jailbreaking, this research is a reminder that the BootROM still occasionally has a surprise left to give.

At Paradigm Shift, we believe that advancing security requires both defensive engineering and deep offensive research. Projects such as usbliter8 help uncover the practical limitations of existing mitigations, improve our understanding of complex attack surfaces, and ultimately contribute to building more resilient systems.