
EMBEDDED HACKING

Reverse Engineering the Raspberry Pi Pico 2

Week 2: Hello, World - Debugging and Hacking Basics: Debugging and Hacking a Basic Program for the Pico 2

ARM Cortex-M33 • GDB • Ghidra

August 14, 2026

For educational and defensive security research only

LEGAL DISCLAIMER: The information, tools, and code provided in this repository and course are strictly for educational, research, and defensive purposes only.

You are explicitly prohibited from using any materials contained herein to access, test, modify, or exploit any device, network, or system that you do not own 100% or for which you do not have explicit, documented, and legally binding authorization to interact with.

By using this repository and course, you acknowledge and agree that:

1. Any illegal, unauthorized, or malicious use of this information is solely your responsibility.
2. The author(s) and contributor(s) of this repository and course shall not be held liable for any damages, legal repercussions, criminal charges, or unauthorized actions resulting from the use, misuse, or abuse of the contents herein.
3. You will comply with all applicable local, state, national, and international laws regarding cybersecurity and computer fraud.

IF YOU DO NOT AGREE WITH THESE TERMS, DO NOT USE THIS REPOSITORY AND COURSE.

What You'll Learn This Week

By the end of this tutorial, you will be able to:

- Connect to a live embedded system using OpenOCD and GDB
- Step through code instruction by instruction and watch the stack change
- Examine memory, registers, and decode little-endian values
- Set strategic breakpoints to pause execution at key moments
- Understand why direct string assignment fails in bare-metal systems
- Write custom data directly to SRAM memory
- Hijack register values to redirect program behavior
- Modify a running program's output in real-time

Review from Week 1

This week builds directly on Week 1 concepts. You should already be comfortable with:

- **Registers** (r0-r12, SP, LR, PC) - We'll watch them change and manipulate r0 to change program behavior
 - **Memory Layout** (Flash at 0x10000000, RAM at 0x20000000) - Critical for understanding where we can write
 - **The Stack** and how push/pop work - We'll watch this in action
 - **Little-Endian** byte ordering - We'll decode values live
 - **GDB basics** from Week 1's dynamic analysis section
 - **Ghidra basics** from Week 1's static analysis section
-

Part 1: Understanding Live Hacking

What is Live Hacking?

Live hacking means modifying a program *while it's running* on real hardware. Instead of changing the source code and recompiling, we intercept the program mid-execution and change what it does on the fly.

Think of it like this: imagine a train is heading to New York City. Live hacking is like switching the tracks while the train is moving so it goes to Los Angeles instead!

Why is This Important?

Live hacking techniques are used for:

- **Security Research:** Finding vulnerabilities in embedded systems
- **Penetration Testing:** Testing if systems can be compromised
- **Malware Analysis:** Understanding how malicious code works
- **Debugging:** Fixing bugs in systems that can't be easily reprogrammed

Real-World Application

“With great power comes great responsibility!”

Imagine you're a security researcher testing an industrial control system at a power plant. You need to verify that an attacker couldn't:

1. Change the values being displayed to engineers
2. Make dangerous equipment appear safe
3. Hide malicious activity from monitoring systems

The techniques you'll learn today are *exactly* how this would be done. Understanding these attacks helps us build better defenses!

Part 2: Review - Memory Layout (from Week 1)

REVIEW: In Week 1, we learned about the RP2350's memory layout. This knowledge is essential for our hack!

Before we hack, let's remember where things live in memory on the RP2350:

The Code We're Hacking

Remember our `0x0001_hello-world.c` program from Week 1:

```
#include <stdio.h>
#include "pico/stdlib.h"

int main(void) {
    stdio_init_all();

    while (true)
        printf("hello, world\r\n");
}
```

This simple program:

1. Initializes I/O with `stdio_init_all()`
2. Enters an infinite `while(true)` loop
3. Prints `"hello, world\r\n"` forever

Our goal: **Make it print something else WITHOUT changing the source code!**

Memory Map

```
+-----+
| Flash Memory (XIP) - READ ONLY |
| Starts at: 0x10000000           |
| Contains: Program code, constant strings |
| NOTE: We CANNOT write to flash during runtime! |
+-----+
| SRAM - READ/WRITE              |
| Starts at: 0x20000000           |
+-----+
```

```
| Contains: Stack, Heap, Variables |
| NOTE: We CAN write to SRAM during runtime! |
+-----+
```

REVIEW: In Week 1, we saw SP values in the `0x20081xxx` range (for example `0x20081fc8`) - that's in the SRAM region. In this run you may see values like `0x20081ff8` depending on where execution is paused. The stack "grows downward" from the top of SRAM.

Why This Matters for Our Hack

The string "hello, world" is stored in **flash memory** (around `0x100019cc`). Flash memory is **read-only** during normal operation - we can't just overwrite it.

But SRAM (starting at `0x20000000`) is **read-write**! This is where we'll create our hacked string.

Part 3: The Attack Plan

Here's our step-by-step attack strategy:

```
+-----+
| STEP 1: Start the debug server (OpenOCD) |
+-----+
| STEP 2: Connect with GDB and halt the program |
+-----+
| STEP 3: Set a breakpoint right before puts() |
+-----+
| STEP 4: When we hit the breakpoint, r0 contains |
|         the address of "hello, world" |
+-----+
| STEP 5: Create our malicious string in SRAM |
+-----+
| STEP 6: Change r0 to point to OUR string |
+-----+
| STEP 7: Continue execution - HACKED! |
+-----+
```

Part 4: Setting Up Your Environment

Prerequisites

Before we start, make sure you have:

1. A Raspberry Pi Pico 2 board with debug probe connected
2. OpenOCD installed and configured
3. GDB (arm-none-eabi-gdb) installed
4. A serial monitor application (like PuTTY, minicom, or screen)
5. The "hello-world" binary loaded on your Pico 2

What You'll Need Open

You will need **THREE** terminal windows:

1. **Terminal 1:** Running OpenOCD (the debug server)
2. **Terminal 2:** Running GDB (where we do the hacking)
3. **PuTTY:** Running your serial monitor (to see output)

Part 5: GDB Deep Dive - Exploring the Binary

Before we start hacking, let's use GDB to thoroughly understand our program. This hands-on tutorial will teach you to examine memory, step through code, and watch the stack in action.

Starting the Debug Environment

Step 0a: Start OpenOCD (Terminal 1)

OpenOCD is the bridge between your computer and the Pico 2's debug interface. It creates a server that GDB can connect to.

Open Terminal 1 and type:

```
openocd -s "$env:USERPROFILE\.pico-sdk\openocd\0.12.0+dev\scripts" -f interface/cmsis-dap.cfg -f target/rp2350.cfg -c
  ↪ "adapter speed 5000"
```

What this command means:

- openocd = the OpenOCD program
- -s ... = path to OpenOCD scripts folder
- -f interface/cmsis-dap.cfg = use the CMSIS-DAP debug probe configuration
- -f target/rp2350.cfg = configure for the RP2350 chip
- -c "adapter speed 5000" = set the debug speed to 5000 kHz

You should see output like:

```
Open On-Chip Debugger 0.12.0
Licensed under GNU GPL v2
Info : Listening on port 3333 for gdb connections
Info : CMSIS-DAP: SWD supported
Info : CMSIS-DAP: Interface ready
```

Important: Leave this terminal running! Don't close it.

Step 0b: Start Your Serial Monitor - PuTTY (Terminal 2)

PuTTY will show us the output from our Pico 2. When we hack the program, we'll see the results here!

To set up PuTTY:

1. Open **PuTTY**
2. In the **Session** category:
 - **Connection type:** Select **Serial**
 - **Serial line:** Enter your COM port (e.g., COM3 - check Device Manager to find yours)
 - **Speed:** Enter 115200
3. Click **Open**

Tip: Finding your COM port: Open Device Manager -> Ports (COM & LPT) -> Look for "USB Serial Device" or "Pico" - note the COM number.

You should see:

```
hello, world
hello, world
hello, world
hello, world
...
```

The program is running and printing "hello, world" in an infinite loop!

Important: Leave PuTTY running! We'll watch it change when we hack the system.

Step 0c: Start GDB (Terminal 3)

Open Terminal 3 and start GDB with your binary:

```
arm-none-eabi-gdb build\0x0001_hello-world.elf
```

What this command means:

- arm-none-eabi-gdb = the ARM version of GDB
- build\0x0001_hello-world.elf = our compiled program with debug symbols

You should see:

```
GNU gdb (Arm GNU Toolchain 13.2) 13.2
Reading symbols from build\0x0001_hello-world.elf...
(gdb)
```

The (gdb) prompt means GDB is ready for commands!

Step 0d: Connect GDB to OpenOCD

Now we need to connect GDB to OpenOCD. OpenOCD is listening on port 3333.

Type this command:

```
(gdb) target extended-remote :3333
```

You should see:

```
Remote debugging using :3333
main ()
    at C:/Users/assem.KEVINTHOMAS/OneDrive/Documents/Embedded-Hacking/0x0001_hello-world/0x0001_hello-world.c:5
5      stdio_init_all();
```

We're connected! GDB shows us the program is currently in the main function.

Step 0e: Halt the Running Program

The program is still running (you can see “hello, world” still printing in PuTTY). Let's stop it:

Type this command:

```
(gdb) monitor reset halt
```

What this command means:

- monitor = send a command to OpenOCD (not GDB)
- reset = reset the processor
- halt = stop execution immediately

You should see:

```
[rp2350.cm0] halted due to debug-request, current mode: Thread
xPSR: 0xf9000000 pc: 0x00000088 msp: 0xf0000000
[rp2350.cm1] halted due to debug-request, current mode: Thread
xPSR: 0xf9000000 pc: 0x00000088 msp: 0xf0000000
```

Check PuTTY: The “hello, world” messages should have stopped! The processor is now frozen, waiting for our commands.

Exploring the Binary

Now that we're connected and the processor is halted, let's explore!

Step 1: Examine Memory Starting at XIP Base Address

Our program code starts at address 0x10000000. Let's look at the first 1000 instructions to find our main function.

Type this command in GDB:

```
(gdb) x/1000i 0x10000000
```

What this command means:

- x = examine memory
- /1000i = show 1000 instructions
- 0x10000000 = starting address

What you're looking for:

Scroll through the output and look for something like this:

```
0x10000234 <main>: push {r3, lr}
```

The <main> label tells us we found our main function! The address 0x10000234 is where main starts.

Step 2: Examine the Main Function in Detail

Now let's look at just the main function. We'll examine 5 instructions starting at the main function address:

Type this command:

```
(gdb) x/5i 0x10000234
```

You should see:

```
(gdb) x/5i 0x10000234
=> 0x10000234 <main>:  push    {r3, lr}
0x10000236 <main+2>:  bl      0x1000156c <stdio_init_all>
0x1000023a <main+6>:  ldr     r0, [pc, #8]    @ (0x10000244 <main+16>)
0x1000023c <main+8>:  bl      0x100015fc <__wrap_puts>
0x10000240 <main+12>: b.n     0x1000023a <main+6>
```

Let's understand each instruction:

Address	Instruction	What It Does
0x10000234	push {r3, lr}	Save r3 and the return address onto the stack
0x10000236	bl 0x1000156c <stdio_init_all>	Call the <code>stdio_init_all</code> function
0x1000023a	ldr r0, [pc, #8]	Load the address of our string into r0
0x1000023c	bl 0x100015fc <__wrap_puts>	Call puts to print our string
0x10000240	b.n 0x1000023a	Jump back to the ldr instruction (infinite loop!)

Step 3: Set a Breakpoint at Main

A **breakpoint** is like a stop sign for your program. When the processor reaches that address, it will pause and let us examine things.

Type this command:

```
(gdb) b *0x10000234
```

You should see:

```
Breakpoint 1 at 0x10000234: file C:/Users/.../0x0001_hello-world.c, line 5.
Note: automatically using hardware breakpoints for read-only addresses.
```

What this means:

- b = set breakpoint
- *0x10000234 = at this exact memory address
- GDB confirms the breakpoint is set and even tells us which line of C code this corresponds to!

Step 4: Continue Execution Until Breakpoint

Now let's run the program until it hits our breakpoint:

Type this command:

```
(gdb) c
```

You should see:

```
Continuing.

Thread 1 "rp2350.cm0" hit Breakpoint 1, main ()
    at C:/Users/.../0x0001_hello-world.c:5
5      stdio_init_all();
```

What happened:

- The processor ran until it reached address `0x10000234`
- It stopped right before executing the instruction at that address
- GDB shows us we're at line 5 of our C source code

Step 5: Examine Instructions with Arrow

Let's look at our instructions again:

Type this command:

```
(gdb) x/5i 0x10000234
```

You should see:

```
(gdb) x/5i 0x10000234
=> 0x10000234 <main>:  push    {r3, lr}
0x10000236 <main+2>:  bl      0x1000156c <stdio_init_all>
0x1000023a <main+6>:  ldr     r0, [pc, #8] @ (0x10000244 <main+16>)
0x1000023c <main+8>:  bl      0x100015fc <__wrap_puts>
0x10000240 <main+12>: b.n     0x1000023a <main+6>
```

Notice the arrow =>! This arrow shows which instruction we're about to execute. We haven't executed it yet - we're paused right before it.

Understanding the Stack in Action**Step 6: Examine the Stack Before Push**

Before we execute the push instruction, let's see what's on the stack:

Type this command:

```
(gdb) x/10x $sp
```

What this command means:

- `x` = examine memory
- `/10x` = show 10 values in hexadecimal
- `$sp` = starting at the stack pointer address

You should see:

```
0x20082000: 0x00000000 0x00000000 0x00000000 0x00000000
0x20082010: 0x00000000 0x00000000 0x00000000 0x00000000
0x20082020: 0x00000000 0x00000000
```

What this shows:

- The stack pointer is at address `0x20082000`
- The stack is empty (all zeros)

- This is the “top” of our stack in RAM

Verify where this value came from (the Vector Table):

Now let’s trace back where this initial `0x20082000` value came from. It comes from the first entry in the vector table at `0x10000000`:

```
(gdb) x/x $sp
0x20082000:      0x00000000
(gdb) x/x 0x10000000
0x10000000 <__vectors>: 0x20082000
```

What this shows:

- The first command `x/x $sp` reads one word at the stack pointer (currently `0x00000000`)
- The second command `x/x 0x10000000` reads the **first entry in the vector table** at address `0x10000000`
- That vector table entry contains `0x20082000` - this is the **initial stack pointer value!**
- The Boot ROM reads this value and puts it into the SP register when the chip starts
- This is why our SP register already has `0x20082000` before `main()` runs

Step 7: Execute One Instruction (Step Into)

Now let’s execute just ONE assembly instruction:

Type this command:

```
(gdb) si
```

What this command means:

- `si` = step instruction (execute one assembly instruction)

You should see:

```
0x10000236 5      stdio_init_all();
```

Let’s verify where we are:

Type this command:

```
(gdb) x/5i 0x10000234
```

You should see:

```
(gdb) x/5i 0x10000234
0x10000234 <main>:  push    {r3, lr}
=> 0x10000236 <main+2>: bl      0x1000156c <stdio_init_all>
0x1000023a <main+6>:  ldr     r0, [pc, #8]    @ (0x10000244 <main+16>)
0x1000023c <main+8>:  bl      0x100015fc <__wrap_puts>
0x10000240 <main+12>: b.n    0x1000023a <main+6>
```

Notice: The arrow `=>` has moved! We’ve executed the push instruction and are now about to execute the `bl` (branch with link) instruction.

Step 8: Examine the Stack After Push

Now let’s see what the push instruction did to our stack:

Type this command:

```
(gdb) x/10x $sp
```

You should see:

```
0x20081ff8: 0xe000ed08 0x1000018f 0x00000000 0x00000000
0x20082008: 0x00000000 0x00000000 0x00000000 0x00000000
0x20082018: 0x00000000 0x00000000
```

What changed:

- The stack pointer moved from `0x20082000` to `0x20081ff8`
- That's 8 bytes lower ($2 * 4\text{-byte values}$)
- Two new values appeared: `0xe000ed08` and `0x1000018f`

Step 9: Verify What Was Pushed

Let's prove that these values came from `r3` and `lr`:

Check `r3`:

```
(gdb) x/x $r3
```

You should see:

```
0xe000ed08: Cannot access memory at address 0xe000ed08
```

This error is expected! The value `0xe000ed08` is in `r3`, and when we try to examine it as an address, that memory location isn't accessible. But we can see the value matches what's on the stack!

Check `lr` (Link Register):

```
(gdb) x/x $lr
```

You should see:

```
0x1000018f <platform_entry+8>: 0x00478849
```

The value `0x1000018f` is in `lr` - this is the return address! This matches the second value on our stack.

Step 10: Verify Stack Layout

Let's look at each pushed value individually:

First pushed value (`r3`):

```
(gdb) x/x $sp
```

You should see:

```
0x20081ff8: 0xe000ed08
```

This is the value from `r3`, pushed first.

Second pushed value (`lr`):

```
(gdb) x/x $sp+4
```

You should see:

```
0x20081ffc: 0x1000018f
```

This is the value from `lr` (the return address), pushed second.

Understanding the Stack Diagram

Before push { <code>r3</code> , <code>lr</code> }:		After push { <code>r3</code> , <code>lr</code> }:	
Address	Value	Address	Value
-----		-----	
0x20082000	(empty) ← SP	0x20082000	(old SP location)
		0x20081ffc	0x1000018f (<code>lr</code>)
		0x20081ff8	0xe000ed08 (<code>r3</code>) ← SP

Key Points:

1. The stack grows **DOWNWARD** (addresses get smaller)
2. The SP always points to the last item pushed
3. `r3` was pushed first, then `lr` was pushed on top of it

Continuing Through the Program

Step 11: Step Over the `stdio_init_all` Function

We don't need to examine every instruction inside `stdio_init_all` - it's just setup code. Let's "step over" it:

First, verify where we are:

```
(gdb) x/5i 0x10000234
```

You should see:

```
(gdb) x/5i 0x10000234
0x10000234 <main>:  push    {r3, lr}
=> 0x10000236 <main+2>: bl      0x1000156c <stdio_init_all>
0x1000023a <main+6>:  ldr     r0, [pc, #8]    @ (0x10000244 <main+16>)
0x1000023c <main+8>:  bl      0x100015fc <__wrap_puts>
0x10000240 <main+12>: b.n     0x1000023a <main+6>
```

Now step over the function call:

```
(gdb) n
```

What this command means:

- `n` = next (step over function calls, don't go inside them)

You should see:

```
8      printf("hello, world\r\n");
```

Verify where we are now:

```
(gdb) x/5i 0x10000234
```

You should see:

```
(gdb) x/5i 0x10000234
0x10000234 <main>:  push    {r3, lr}
0x10000236 <main+2>: bl      0x1000156c <stdio_init_all>
=> 0x1000023a <main+6>:  ldr     r0, [pc, #8]    @ (0x10000244 <main+16>)
0x1000023c <main+8>:  bl      0x100015fc <__wrap_puts>
0x10000240 <main+12>: b.n     0x1000023a <main+6>
```

The arrow has moved past the function call!

Understanding the LDR Instruction

We're now at:

```
ldr r0, [pc, #8] @ (0x10000244 <main+16>)
```

What does this instruction do?

1. Take the current Program Counter (PC) value
2. Add 8 to it
3. Go to that memory address (0x10000244)
4. Load the value stored there into `r0`

This is loading a **pointer** - the address of our "hello, world" string!

Step 13: Execute the LDR and Examine `r0`

Execute one instruction:

```
(gdb) si
```

You should see:

```
0x1000023c  8      printf("hello, world\r\n");
```

Now examine what's in r0:

```
(gdb) x/x $r0
```

You should see:

```
0x100019cc:  0x6c6c6568
```

Step 14: Decoding the Mystery Value

The value 0x6c6c6568 looks strange, but it's actually ASCII characters! Let's decode it:

ASCII Table Reference:

Hex	Character
0x68	h
0x65	e
0x6c	l
0x6c	l

So 0x6c6c6568 = "leh" backwards!

Why is it backwards?

This is called **little-endian** byte order. The RP2350 stores bytes in reverse order in memory. When we read them as a 32-bit value, they appear reversed.

Step 15: View the Full String

Let's tell GDB to show this as a string instead of a hex number:

```
(gdb) x/s $r0
```

What this command means:

- x = examine memory
- /s = show as a string
- \$r0 = at the address stored in r0

You should see:

```
0x100019cc: "hello, world\r"
```

There's our string! The \r is a carriage return character (part of \r\n).

Key Discovery: The string "hello, world" is stored at address 0x100019cc in flash memory. This is the value that gets loaded into r0 before calling puts(). We'll use this knowledge in our hack!

Part 6: Continuing the Debug Session for the Hack

You're right where you need to be from Part 5, so we **do not restart** OpenOCD or GDB here.

Use this as a quick checkpoint before the hack:

- OpenOCD is still running and listening on :3333
- GDB is still connected (target extended-remote :3333 already done)
- The target is halted at a known point (or easy to re-halt)

If your session is still live, continue directly to Part 7.

If you got disconnected, run only this minimal recovery in GDB:

```
(gdb) target extended-remote :3333
(gdb) monitor reset halt
```

After that, continue to the analysis steps below.

Part 7: Analyzing the Target

REVIEW: We're using the same GDB commands we learned earlier. The `x` command examines memory, and `/5i` shows 5 instructions.

Step 6: Examine the Main Function

Let's look at the main function to understand what we're dealing with:

Type this command:

```
(gdb) x/5i 0x10000234
```

What this command means:

- `x` = examine memory (Week 1 review!)
- `/5i` = show 5 instructions
- `0x10000234` = the address of main (we found this in Week 1!)

You should see:

```
(gdb) x/5i 0x10000234
0x10000234 <main>:  push    {r3, lr}
0x10000236 <main+2>:  bl      0x1000156c <stdio_init_all>
0x1000023a <main+6>:  ldr     r0, [pc, #8]    @ (0x10000244 <main+16>)
=> 0x1000023c <main+8>:  bl      0x100015fc <__wrap_puts>
0x10000240 <main+12>:  b.n     0x1000023a <main+6>
```

REVIEW: This is the same disassembly we analyzed in Week 1! Remember:

- `push {r3, lr}` saves registers to the stack
- `bl` is "branch with link" - it calls a function and saves the return address in LR
- `b.n` is the infinite loop that jumps back to the `ldr` instruction

Understanding the Code Flow

REVIEW: In Week 1, we learned that `r0-r3` are used to pass arguments to functions. The first argument always goes in `r0`!

Let's break down what happens each time through the loop:

Address	Instruction	What Happens
0x1000023a	<code>ldr r0, [pc, #8]</code>	Load the address of "hello, world" into <code>r0</code>
0x1000023c	<code>bl __wrap_puts</code>	Call <code>puts()</code> - it reads the string address from <code>r0</code> !

Address	Instruction	What Happens
0x10000240	b.n 0x1000023a	Jump back to the ldr instruction (loop forever)

How This Maps to Our C Code

```
while (true)
    printf("hello, world\r\n"); // The compiler optimized this to puts()
```

The compiler:

1. Loads the string address into r0 (first argument)
2. Calls puts() (optimized from printf() since we're just printing a string)
3. Loops back forever with b.n

The Key Insight: Right before the bl __wrap_puts instruction (at 0x1000023c), the register r0 contains the address of the string to print!

If we can change what r0 points to, we can make it print **anything we want!**

Part 8: Setting the Trap

REVIEW: In Week 1, we used b main and b *0x10000234 to set breakpoints. Now we'll use the same technique at a more strategic location!

Step 7: Set a Strategic Breakpoint

We want to stop the program RIGHT BEFORE it calls puts(). That's at address 0x1000023c.

Type this command:

```
(gdb) b *0x1000023c
```

What this command means:

- b = set a breakpoint (same as Week 1!)
- *0x1000023c = at this exact memory address (the asterisk means "address")

You should see:

```
Breakpoint 2 at 0x1000023c: file
↳ C:/Users/assem.KEVINTHOMAS/OneDrive/Documents/Embedded-Hacking/0x0001_hello-world/0x0001_hello-world.c, line 8
```

What does "hardware breakpoints" mean?

Because our code is in flash memory (read-only), GDB can't insert a software breakpoint by modifying the code. Instead, it uses a special feature of the ARM processor called a **hardware breakpoint**. The processor has a limited number of these (usually 4-8), but they work on any memory type.

Step 8: Continue Execution and Hit the Breakpoint

Now let's run the program until it hits our breakpoint:

Type this command:

```
(gdb) c
```

What this command means:

- c = continue (run until something stops us)

You should see:

Continuing.

```
Thread 1 "rp2350.cm0" hit Breakpoint 2, 0x1000023c in main ()
    at C:/Users/assem.KEVINTHOMAS/OneDrive/Documents/Embedded-Hacking/0x0001_hello-world/0x0001_hello-world.c:8
8         printf("hello, world\r\n");
```

The program has stopped RIGHT BEFORE calling puts()! The string address is loaded into r0, but the function hasn't been called yet.

Step 9: Verify Our Position with Disassembly

Let's double-check where we are using the disas command:

Type this command:

```
(gdb) disas
```

What this command means:

- disas = disassemble the current function

You should see:

```
(gdb) disas
Dump of assembler code for function main:
0x10000234 <+0>:    push    {r3, lr}
0x10000236 <+2>:    bl      0x1000156c <stdio_init_all>
0x1000023a <+6>:    ldr     r0, [pc, #8]    @ (0x10000244 <main+16>)
=> 0x1000023c <+8>:    bl      0x100015fc <__wrap_puts>
0x10000240 <+12>:   b.n     0x1000023a <main+6>
0x10000242 <+14>:   nop
0x10000244 <+16>:   adds    r4, r1, r7
0x10000246 <+18>:   asrs    r0, r0, #32
```

Notice the arrow => pointing to 0x1000023c! This confirms we're about to execute the bl __wrap_puts instruction. Perfect!

Part 9: Examining the Current State

REVIEW: In Week 1, we used x/s \$r0 to view the "hello, world" string. We also learned about **little-endian** byte ordering - remember how 0x6c6c6568 spelled "lleh" backwards?

Step 10: Examine What's in r0

Let's see what string r0 is currently pointing to:

Type this command:

```
(gdb) x/s $r0
```

What this command means:

- x = examine memory (Week 1 review!)
- /s = display as a string
- \$r0 = the address stored in register r0

You should see:

```
0x100019cc:    "hello, world\r"
```

There it is! The register r0 contains 0x100019cc, which is the address of our "hello, world" string in flash memory.

REVIEW: This is the same address `0x100019cc` we discovered in Week 1, Step 15 when we used `x/s $r0` after executing the `ldr` instruction!

Part 10: The Failed Hack Attempt (Learning Why)

Step 11: Try to Directly Change the String (This Will Fail!)

Your first instinct might be to just assign a new string to `r0`. Let's try it and see what happens:

Type this command:

```
(gdb) set $r0 = "hacky, world\r"
```

You should see an error:

```
evaluation of this expression requires the program to have a function "malloc".
```

Oh no! It didn't work!

Why Did This Fail?

This is a very important lesson! Here's what happened:

1. When you type `"hacky, world\r"` in GDB, GDB interprets this as: "Create a new string and give me its address"
2. To create a new string at runtime, GDB tries to allocate target memory using `malloc()`.
3. If the target program exposed a working `malloc()` that GDB could call, this style of assignment can work.
4. In our case, this is a bare-metal firmware image and there is no usable `malloc()` path for GDB expression evaluation, so GDB cannot create temporary storage for that string literal.
5. That is why this command fails here, and why we switch to writing bytes directly into SRAM ourselves.

Let's verify nothing changed:

```
(gdb) x/s $r0
```

You should see:

```
0x100019cc:  "hello, world\r"
```

The original string is still there. Our hack attempt failed... but we're not giving up!

Part 11: The Real Hack - Writing to SRAM

Step 12: Understanding the Solution

Since we can't use `malloc()`, we need to manually create our string somewhere in memory. Remember our memory map?

- Flash (`0x10000000`): **Read-only** - can't write here
- SRAM (`0x20000000`): **Read-write** - we CAN write here!

REVIEW: In Week 1 we observed SP in the `0x20081xxx` range (for example `0x20081fc8`). In this run, after `push {r3, lr}`, you are seeing `0x20081ff8`, which is also correct and still near the top of SRAM. We'll write our string at a safer SRAM address (`0x20040000`) to avoid vector-table and stack conflicts.

We'll write our malicious string directly to SRAM, then point `r0` to it.

Step 13: Create Our Malicious String in SRAM

We need to write 13 bytes (12 characters + null terminator) to SRAM:

Character	ASCII Hex
h	-
a	-
c	-
k	-
y	-
,	-
(space)	-
w	-
o	-
r	-
l	-
d	-
\r	-
\0	-

Type this command:

```
(gdb) set {char[13]} 0x20040000 = "hacky, world"
```

What this command means:

- set = modify memory
- {char[13]} = treat the target as an array of 13 characters
- 0x20040000 = the address where we're writing (safe SRAM offset)
- = "hacky, world" = the string bytes to write

No output means success!

Step 14: Verify Our String Was Written

Let's confirm our malicious string is in SRAM:

Type this command:

```
(gdb) x/s 0x20040000
```

You should see:

```
0x20040000: "hacky, world"
```

OUR STRING IS IN MEMORY!

The important thing is our string is in writable SRAM at a safe offset and ready to use.

Part 12: Hijacking the Register

REVIEW: In Week 1, we learned that `r0` holds the first argument to a function. When `puts()` is called, it expects `r0` to contain a pointer to the string it should print. By changing `r0`, we change what gets printed!

Step 15: Change `r0` to Point to Our String

Now for the magic moment! We'll change `r0` from pointing to the original string to pointing to OUR string:
Type this command:

```
(gdb) set $r0 = 0x20040000
```

What this command means:

- `set` = modify a value
- `$r0` = the `r0` register
- `= 0x20040000` = change it to this address (where our string is)

No output means success!

Step 16: Verify the Register Was Changed

Let's confirm `r0` now points to our malicious string:

First, check one byte (explicit byte view):

```
(gdb) x/bx $r0
```

You should see:

```
0x20040000: 0x68
```

The value `0x68` is ASCII 'h', the first byte of "hacky, world".

Now check one 32-bit word (explicit word view):

```
(gdb) x/wx $r0
```

You should see:

```
0x20040000: 0x6b636168
```

This is the first 4 bytes (h a c k) packed into one 32-bit little-endian word.

Now check it as a string:

```
(gdb) x/s $r0
```

You should see:

```
0x20040000: "hacky, world"
```

THE HIJACK IS COMPLETE! When `puts()` runs, it will read the string address from `r0` - which now points to our malicious string!

Part 13: Executing the Hack

Step 17: Continue Execution

This is the moment of truth! Let's continue the program and watch our hack take effect:

Type this command:

```
(gdb) c
```

You should see:

```
Continuing.

Thread 1 "rp2350.cm0" hit Breakpoint 2, 0x1000023c in main ()
    at C:/Users/assem.KEVINTHOMAS/OneDrive/Documents/Embedded-Hacking/0x0001_hello-world/0x0001_hello-world.c:8
8      printf("hello, world\r\n");
```

The program ran through one loop iteration and hit our breakpoint again.

Step 18: Check Your Serial Monitor!

Look at Terminal 2 (your serial monitor)!

You should see:

```
hello, world
hello, world
hello, world
hacky, world    <--- OUR HACK!
```

BOOM! WE DID IT!

You just modified a running program on real hardware! The processor executed code that was supposed to print “hello, world” but instead printed “hacky, world” because we hijacked the data it was using!

Part 14: Static Analysis with Ghidra - Understanding the Hack

Now that we’ve performed the hack dynamically with GDB, let’s use Ghidra to understand the same concepts through static analysis. This shows how you could plan such an attack without even connecting to the hardware!

Opening the Project in Ghidra

If you haven’t already set up the Ghidra project from Week 1:

1. Launch Ghidra
2. Select **File -> New Project -> Non-Shared Project**
3. Name it 0x0001_hello-world
4. Drag and drop 0x0001_hello-world.elf into the project
5. Double-click to open in CodeBrowser
6. Click **Yes** to auto-analyze

Step 1: Navigate to Main

1. In the **Symbol Tree** panel (left side), expand **Functions**
2. Find and click on main

What you’ll see in the Listing View:

```
*****
*                                     FUNCTION
*****
int main (void )
    assume LRset = 0x0
    assume TMode = 0x1
    r0:4          <RETURN>

main
XREF[3]:      Entry Point (*),
              _reset_handler:1000018c (c) ,
              .debug_frame:00000018 (*)

0x0001_hello-world.c:4 (2)
0x0001_hello-world.c:5 (2)
10000234 08 b5      push      {r3,lr}
```

```

0x0001_hello-world.c:5 (4)
10000236 01 f0 99 f9    bl      stdio_init_all          _Bool stdio_init_all(void)
LAB_1000023a                                         XREF[1]: 10000240 (j)
0x0001_hello-world.c:7 (6)
0x0001_hello-world.c:8 (6)
1000023a 02 48      ldr      r0=>__EH_FRAME_BEGIN__ ,[DAT_10000244 ]    = "hello, world\r"
                                         = 100019CCh
1000023c 01 f0 de f9    bl      __wrap_puts          int __wrap_puts(char * s)
0x0001_hello-world.c:7 (8)
10000240 fb e7      b       LAB_1000023a
10000242 00      ??      00h
10000243 bf      ??      BFh
DAT_10000244                                         XREF[1]:  main:1000023a (R)
10000244 cc 19 00 10    undefine 100019CCh          ? -> 100019cc

```

What you'll see in the Decompile View:

```

int main(void)
{
    stdio_init_all();
    do {
        __wrap_puts("hello, world\r");
    } while( true );
}

```

Step 2: Identify the Attack Point

In our GDB hack, we set a breakpoint at 0x1000023c - right before bl __wrap_puts. Let's understand why this was the perfect attack point:

Click on address 0x1000023c in the Listing view.

Notice:

- The instruction is bl __wrap_puts - a function call
- The previous instruction at 0x1000023a loaded r0 with the string address
- Ghidra shows = "hello, world\r" right in the listing!

Key Insight: Ghidra already tells us the string value! In the Listing, you can see = "hello, world\r" and = 100019CCh. This is the exact address we discovered through GDB!

Step 3: Find the String in Memory

Let's trace where the string actually lives:

1. In the Listing view, look at address 0x1000023a:

```

LAB_1000023a                                         XREF[1]: 10000240 (j)
0x0001_hello-world.c:7 (6)
0x0001_hello-world.c:8 (6)
1000023a 02 48      ldr      r0=>__EH_FRAME_BEGIN__ ,[DAT_10000244 ]    = "hello, world\r"
                                         = 100019CCh

```

2. **Double-click on DAT_10000244** to go to the data reference

3. You'll see:

```

DAT_10000244                                         XREF[1]:  main:1000023a (R)
10000244 cc 19 00 10    undefine 100019CCh          ? -> 100019cc

```

4. **Double-click on 100019CCh** to navigate to the actual string

You'll arrive at the string data:

```

//
// .rodata
// SHT_PROGBITS [0x100019cc - 0x10001b17]

```

```

// ram:100019cc-ram:10001b17
//
__init_array_end
__boot2_start__
__boot2_end__
__EH_FRAME_BEGIN__

100019cc 68 65 6c      ds      "hello, world\r"
        6c 6f 2c
        20 77 6f
XREF[5]:  Entry Point (*),
         frame_dummy:10000218 (*),
         main:1000023a (*),
         runtime_init:1000138a (R),
         _elfSectionHeaders:0000005c (*)

```

Step 4: Understand Why We Needed SRAM

Look at the string address: 0x100019cc

This starts with 0x10... which means it's in **Flash memory (XIP region)**!

Address Range	Memory Type	Writable?
0x10000000+	Flash (XIP)	NO - Read Only
0x20000000+	SRAM	YES - Read/Write

This is why our direct string modification failed in GDB! The string lives in flash memory, which is read-only at runtime. We had to create our malicious string in SRAM at a safe address (0x20040000) instead.

Step 5: Examine Cross-References

Ghidra's cross-reference feature shows everywhere a value is used:

1. Navigate back to main (press **G**, type main, press Enter)
2. Click on __wrap_puts at address 0x1000023c
3. Right-click and select ****References -> Show References to __wrap_puts****

This shows every place that calls puts(). In a larger program, you could find ALL the print statements and potentially modify any of them!

Step 6: Use the Decompiler to Plan Attacks

The Decompile view makes attack planning easy:

```

int main(void)
{
    stdio_init_all();
    do {
        __wrap_puts("hello, world\r");
    } while( true );
}

```

From this view, you can immediately see:

- The program loops forever (do { } while (true))
- It calls __wrap_puts() with a string argument
- To change the output, you need to change what's passed to puts()

Step 7: Viewing the String in the .rodata Section

When you navigate to the string address 0x100019cc, you'll see the string stored in the .rodata (read-only data) section:

```

//
// .rodata
// SHT_PROGBITS [0x100019cc - 0x10001b17]
// ram:100019cc-ram:10001b17

```

```

//
__init_array_end
__boot2_start__
__boot2_end__
__EH_FRAME_BEGIN__
XREF[5]:      Entry Point (*),
              frame_dummy:10000218 (*),
              main:1000023a (*),
              runtime_init:1000138a (R),
              _elfSectionHeaders::0000005c (*)

100019cc 68 65 6c      ds      "hello, world\r"
        6c 6f 2c
        20 77 6f

```

This shows the raw bytes of our string: 68 65 6c 6c 6f 2c 20 77 6f... which spell out "hello, world\r" in ASCII.

Step 8: Patching Data in Ghidra (Preview)

Ghidra allows you to modify data directly in the binary! Here's how to patch the string:

1. **Navigate to the string** at address 0x100019cc
2. **Right-click** on the string "hello, world\r" in the Listing view
3. **Select Patch Data** from the context menu
4. **Type** your new string: "hacky, world\r"
5. **Press Enter** to apply the patch

Important: The new string must be the **same length or shorter** than the original! If your new string is longer, it will overwrite adjacent data and likely crash the program.

Original String	Patched String	Result
hello, world\r (14 bytes)	hacky, world\r (14 bytes)	Works perfectly
hello, world\r (14 bytes)	PWNED!\r (7 bytes)	Works (shorter is OK)
hello, world\r (14 bytes)	this is a much longer string\r	Overwrites other data!

After patching, you'll see the change reflected in the Listing view:

```

100019cc 68 61 63      ds      "hacky, world\r"
        6b 79 2c
        20 77 6f

```

Notice how the bytes changed: 68 65 6c 6c 6f ("hello") became 68 61 63 6b 79 ("hacky")!

Looking Ahead: Persistent Binary Patching

Coming in Future Lessons: What we've done in Ghidra so far is just a **preview** of the patch - it modifies the data in Ghidra's view, but doesn't save it back to the actual binary file.

In future lessons, we will learn how to:

1. **Export the patched binary** from Ghidra to create a modified .elf or .bin file
2. **Flash the patched binary** to the Pico 2, making the hack **persistent**

The key difference:

Technique	Persistence	When It's Useful
GDB Live Hacking (this week)	Temporary - lost on reset	Testing, debugging, quick exploitation

Technique	Persistence	When It's Useful
Ghidra Patch Preview (this step)	None - just visualization	Planning and verifying patches
Binary Patching (future lessons)	Permanent - survives reboot	Persistent backdoors, firmware mods

This step helps you understand the mechanics of modifying binary data. Once you're comfortable with this concept, you'll be ready to create truly persistent modifications!

Comparing GDB and Ghidra Approaches

Task	GDB (Dynamic)	Ghidra (Static)
Find main address	x/1000i 0x10000000 + search	Symbol Tree -> Functions -> main
Find string address	Step through <code>ldr</code> , examine <code>\$r0</code>	Click on <code>ldr</code> - shows = 100019CCh
See string content	x/s \$r0	Double-click address -> see ds "hello, world"
Identify attack point	Set breakpoints, step, observe	Read decompiled code, find function calls
Verify memory type	Know address ranges	Check address prefix (0x10... vs 0x20...)

Why Use Both Tools?

- **Ghidra** helps you **plan** the attack by understanding code structure
- **GDB** lets you **execute** the attack and modify live values
- Together, they form a complete reverse engineering workflow!

Ghidra Tips for Attack Planning

1. **Use the Decompiler** - It shows you the high-level logic without decoding assembly
2. **Follow Cross-References** - Find all places a function or variable is used
3. **Check Address Ranges** - Quickly identify Flash vs SRAM locations
4. **Add Comments** - Press ; to annotate what you discover for later
5. **Rename Variables** - Right-click -> Rename to give meaningful names

Part 15: Summary and Review

What We Accomplished

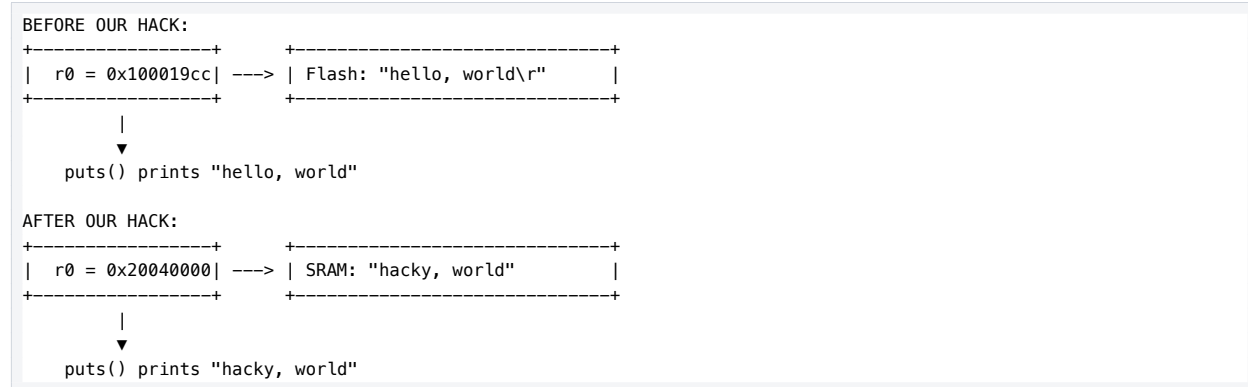
We successfully performed a **live memory injection attack**:

1. **Connected** to a running embedded system using OpenOCD and GDB
2. **Analyzed** the program flow to find the perfect attack point
3. **Set a breakpoint** right before the critical function call
4. **Discovered** that direct string assignment doesn't work without `malloc()`
5. **Wrote** our malicious data directly to SRAM
6. **Hijacked** the `r0` register to point to our data
7. **Executed** the hack and watched the output change!

Week 1 Concepts We Applied

Week 1 Concept	How We Used It This Week
Memory Layout (Flash vs RAM)	We knew flash is read-only, so we wrote to SRAM
Registers (r0)	We hijacked r0 to point to our malicious string
GDB x command	We examined memory and verified our injected string
GDB breakpoints (b)	We set a strategic breakpoint before puts()
Disassembly (disas)	We found the exact instruction to target
Little-endian	We understood how our string bytes are stored

The Attack Flow Diagram



New GDB Commands We Learned

Command	What It Does	New/Review
target extended-remote :3333	Connect to OpenOCD debug server	New
monitor reset halt	Reset and halt the processor	New
disas	Disassemble the current function	Review
x/Ni ADDRESS	Examine N instructions at ADDRESS	Review
x/s ADDRESS	Examine memory as a string	Review
b *ADDRESS	Set breakpoint at exact address	Review
c	Continue execution	Review
set \$r0 = VALUE	Change a register's value	New
set {char[N]} ADDR = {...}	Write characters directly to memory	New

Key Memory Addresses

Address	What's There	Read/Write?
0x10000234	Start of main() function	Read-only
0x1000023c	The bl __wrap_puts call	Read-only
0x100019cc	Original "hello, world" string	Read-only

Address	What's There	Read/Write?
0x20040000	Safe SRAM location (hack target)	Read-Write

Key Takeaways

Building on Week 1

1. **Memory layout knowledge is power** - Understanding that flash is read-only and SRAM is read-write was essential for our hack. This directly built on Week 1's memory map lesson.
2. **Registers control everything** - In Week 1, we watched registers change during execution. This week, we CHANGED them ourselves to alter program behavior.
3. **GDB is a hacking tool** - The same commands we used for learning (x, b, c, disas) are the same commands used for exploitation.

New Concepts

4. **Flash is Read-Only at Runtime** - You can't modify code or constant strings in flash memory while the program runs. You must use SRAM.
 5. **Bare-Metal Means No Runtime** - Without an operating system, there's no malloc(), no dynamic memory allocation. You have to manage memory manually.
 6. **Registers Are the Key** - Function arguments are passed in registers (r0, r1, etc.). By changing these registers at the right moment, you can change what functions do.
 7. **Timing is Everything** - We had to set our breakpoint at exactly the right instruction. One instruction earlier, and r0 wouldn't be loaded yet. One instruction later, and puts() would already have the wrong address.
 8. **This is Real Hacking** - The techniques you learned today are used by security researchers, penetration testers, and yes, attackers. Understanding these attacks helps us build more secure systems.
-

Security Implications

How Would This Work in the Real World?

Imagine an attacker with physical access to an industrial control system:

Scenario	Attack
Nuclear Centrifuge	Change the displayed RPM from dangerous (15,000) to safe (10,000)
Medical Device	Modify dosage readings to hide an overdose
Vehicle ECU	Alter speedometer reading while car actually speeds
Smart Lock	Change the "locked" status to "unlocked"

How Do We Defend Against This?

1. **Disable Debug Ports** - Production devices should have JTAG/SWD disabled
2. **Secure Boot** - Verify firmware hasn't been tampered with
3. **Memory Protection** - Use ARM's MPU to restrict memory access
4. **Tamper Detection** - Hardware that detects physical intrusion
5. **Encryption** - Keep sensitive data encrypted in memory

Glossary

New Terms This Week

Term	Definition
Bare-Metal	Programming directly on hardware without an operating system
CMSIS-DAP	A standard debug interface protocol for ARM processors
Hijack	Taking control of a value or flow that was intended for something else
Hardware Breakpoint	A breakpoint implemented in CPU hardware, works on any memory
Memory Injection	Writing attacker-controlled data into a program's memory space
OpenOCD	Open On-Chip Debugger - software that interfaces with debug hardware
Register Manipulation	Changing the values stored in CPU registers
SRAM	Static Random Access Memory - fast, volatile, read-write memory
Software Breakpoint	A breakpoint implemented by modifying code (requires writable memory)

Review Terms from Week 1

Term	Definition	Where We Used It
Breakpoint	A marker that pauses program execution at a specific location	Part 7 - Setting the trap
Register	Fast storage inside the processor	Part 11 - Hijacking <code>r0</code>
Stack Pointer	Register that points to the top of the stack	Part 2 - Memory layout
XIP	Execute In Place - running code directly from flash	Part 2 - Why we can't write to flash
Little-Endian	Storing the least significant byte at the lowest address	Part 10 - String storage