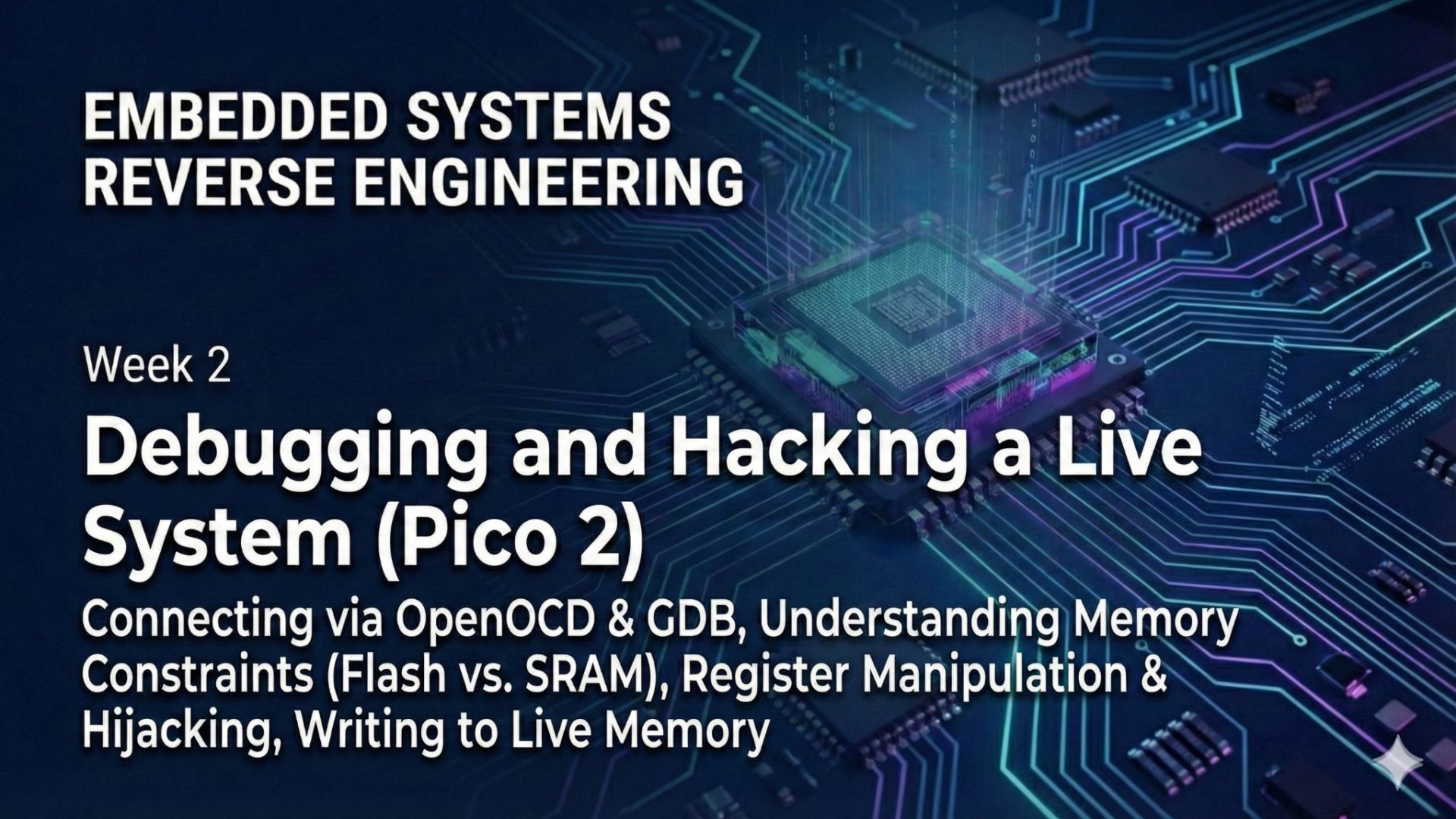


EMBEDDED SYSTEMS
REVERSE ENGINEERING

Week 2

Debugging and Hacking a Live System (Pico 2)

Connecting via OpenOCD & GDB, Understanding Memory Constraints (Flash vs. SRAM), Register Manipulation & Hijacking, Writing to Live Memory

What is Live Hacking?

Definition: Modifying a program's behavior *while* it is executing on real hardware, without recompiling the source code.

Why do we do it?

- **Security Research**: Testing for vulnerabilities in real-time.
- **Malware Analysis**: Watching how malicious code behaves when it thinks no one is looking.
- **Debugging**: Fixing bugs in systems that cannot be easily restarted or reprogrammed.

The Target Program & Goal

The Code: 0x0001_hello-world.c

- Initializes I/O (stdio_init_all).
- Enters an infinite loop.
- Prints “hello, world\r\n” forever.

```c
int main(void) {
  stdio_init_all();
  while (true)
    printf("hello, world\r\n");
}
```

The Mission

- Intercept the program mid-execution.
- Change the output string to “hacky, world”.
- **Constraint:** We cannot modify the source code or recompile. We must manipulate the binary state directly.

Memory Layout Review (Critical for Hacking)

To hack successfully, we must know *where* we can write data.

Flash Memory (XIP)

- **Address:** Starts at 0x10000000
- **Contents:** Program code and constant strings ("hello, world").
- **Property: READ-ONLY** during runtime. We cannot overwrite the string here.

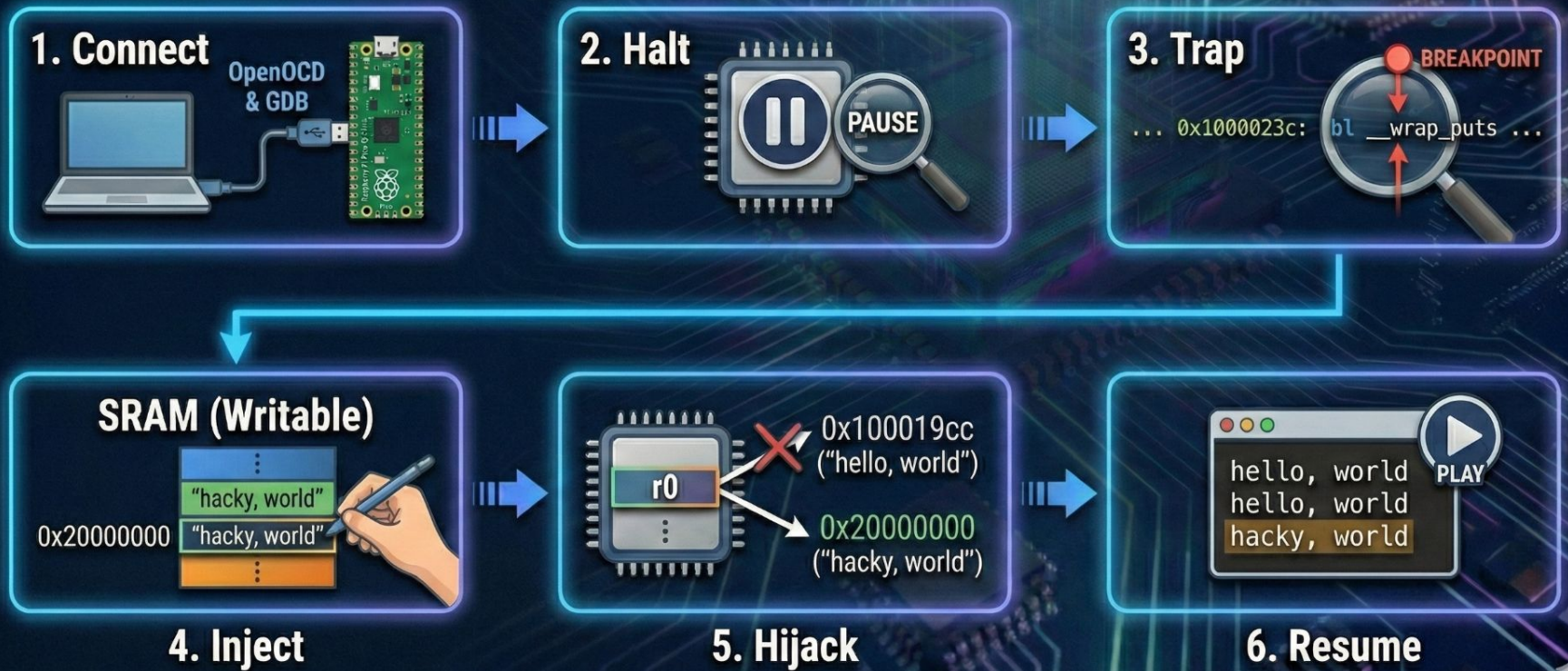
SRAM (Static RAM)

- **Address:** Starts at 0x20000000
- **Contents:** Stack, Heap, Variables.
- **Property: READ-WRITE**. This is our playground for injecting new data.

Constraint: We cann

The Attack Plan

How we will hijack the program execution:



Tool Theory: The Debug Chain

1. OpenOCD (Open On-Chip Debugger)

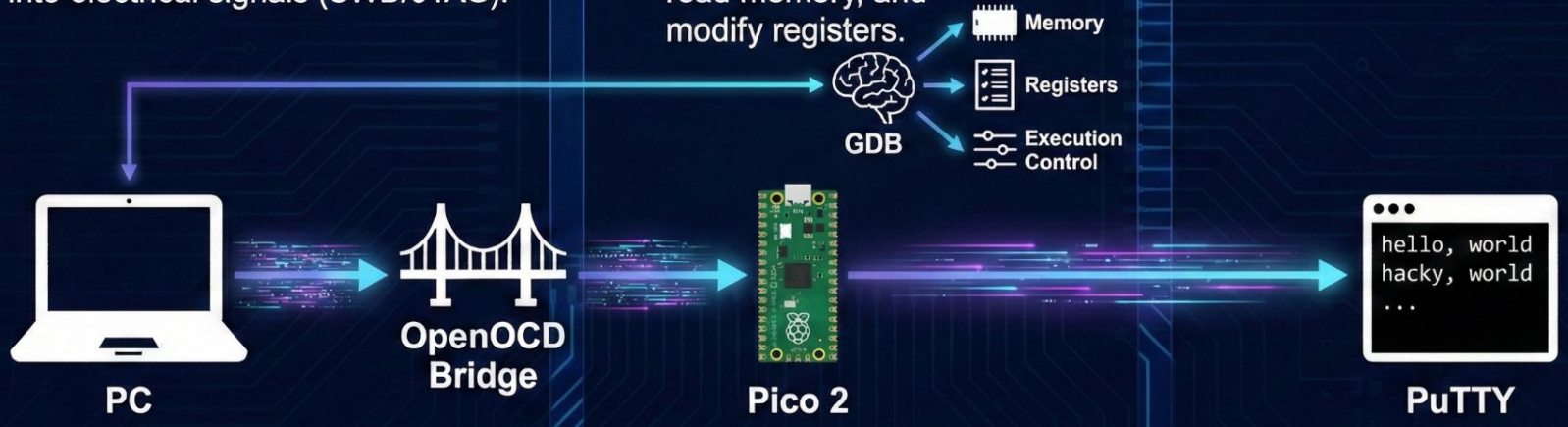
- The bridge between your PC and the Pico 2 hardware.
- Translates high-level commands into electrical signals (SWD/JTAG).

2. GDB (GNU Debugger)

- The “Brain.” Connects to OpenOCD.
- Allows us to stop execution, read memory, and modify registers.

3. Serial Monitor (PuTTY)

- The “Eyes.” Shows us the actual output coming from the Pico 2.



Analyzing the Target (Disassembly)

GDB Disassembly Output (x/5i 0x10000234)

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

Loads address of "hello, world" into r0.

Calls the print function (puts).

The Critical Vulnerability

The function **puts** takes one argument (the string to print). In ARM architecture, the **first argument is always passed in register r0**.

*If we change `r0` at address **0x1000023c**, we change what prints.*

The Failed Attempt (Why `malloc` is missing)

The Mistake & Error

```
(gdb) set $r0 = "hacky, world\r"
evaluation of this expression requires the program to have a function "malloc".
```

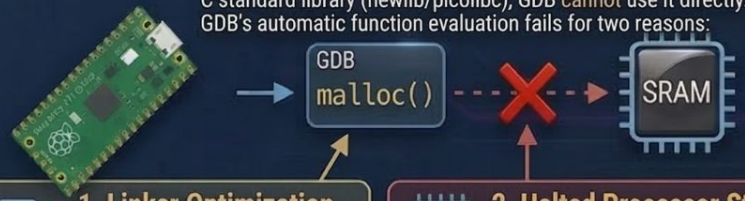
❌ Failed!

Why it fails:

Standard computers have an Operating System (OS) to handle memory allocation (`malloc`).

Microcontrollers (Bare Metal)

Even though the Pico C SDK includes `malloc()` and a complete C standard library (newlib/picolibc), GDB cannot use it directly. GDB's automatic function evaluation fails for two reasons:



1. Linker Optimization (Dead Code Elimination)

(Compiler/ Linker)

The function `malloc()` is removed from the final binary if never explicitly called in your code to save flash memory.

2. Halted Processor State

HALTED

To execute `malloc()`, the processor must run complex code. GDB operates when the core is halted, making dynamic memory calls unsafe and contextually impossible.

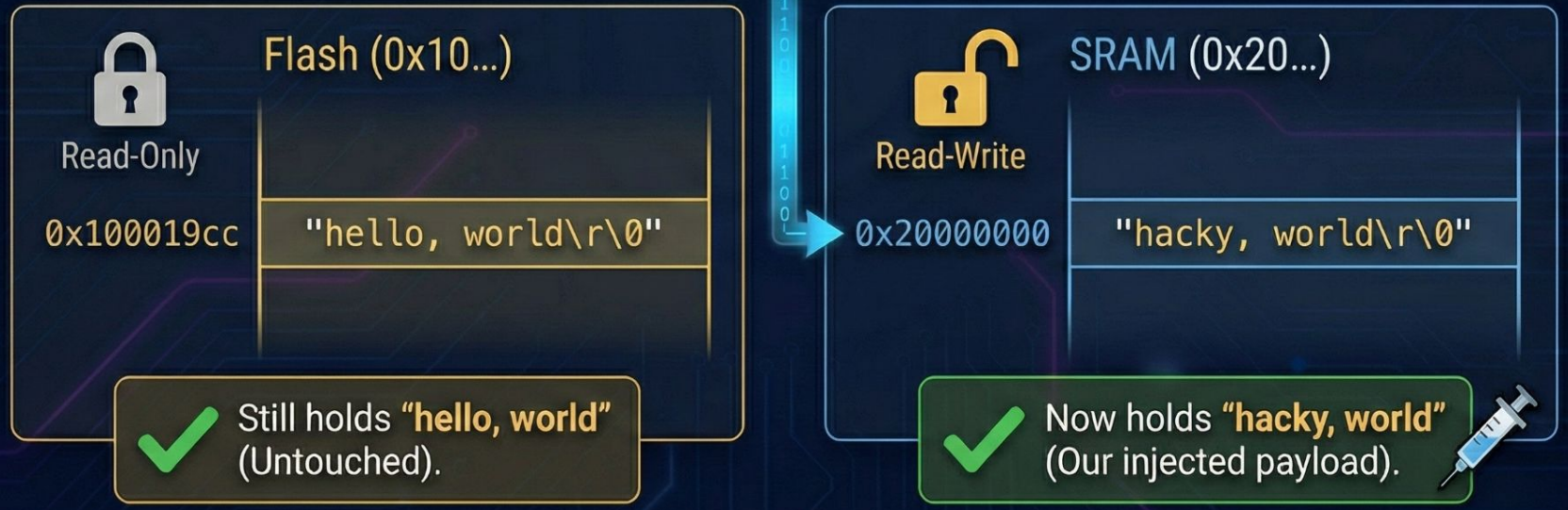
Solution: We must manually act as the memory manager and write bytes directly to a specific address in SRAM.

The Injection (Writing to SRAM)

Since we can't write to Flash (Read-Only), we write to the **base of SRAM** (0x20000000).

```
(gdb) set {char[14]} 0x20000000 = {'h','a','c','k','y',',','', ','w','o','r','l','d','\r','\0'}
```

The Result:



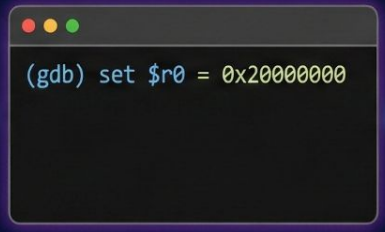
The Hijack (Register Manipulation)

Now that our data is in memory, we must tell the CPU to use it.

Step 1: Paused at Breakpoint

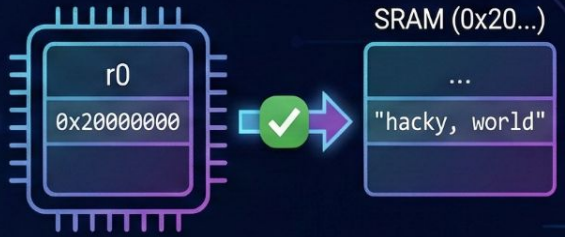
r0 currently points to 0x100019cc (Flash).

Step 2: Execute GDB Command



We execute `set $r0 = 0x20000000`.

Step 3: Register Updated



r0 now points to the start of SRAM.

Execution Flow:

When we type `continue`, the `puts` function looks at `r0`, sees the SRAM address, and prints our hacked string instead of the original.

Key Takeaways for Week 2

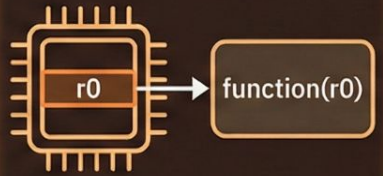
Hardware Constraints Matter

You cannot write to Flash execution memory at runtime; you must use RAM.

Bare Metal Realities

There is no OS to help you. You must manage memory and addresses manually.

Registers are King



Controlling `r0` (the function argument register) gives you control over function behavior.

Static vs. Dynamic

Ghidra (Static)

Helps us _plan_ the attack by finding addresses.

GDB (Dynamic)

Allows us to _execute_ the attack by modifying the live state.