

Embedded Systems Reverse Engineering

// WEEK 02

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

George Mason University

RP2350 // ARM Cortex-M33

Live Hacking Overview

Introduction to Live Hacking

What Is Live Hacking?

Modify a program
WHILE it is running
on real hardware

The Train Analogy

Train heading to NYC
Switch the tracks
while it moves

Now it goes to LA!

Why It Matters

Security research
Penetration testing
Malware analysis
Hardware debugging

No recompile needed!

This Week's Goal

Target Program

hello-world.c
Prints "hello, world"
in infinite loop

Our Mission

Make it print
something ELSE
without changing
the source code

Tools Used

GDB = live debug
OpenOCD = HW bridge
Ghidra = analysis

Hack the running binary

GDB Debug Session

GDB Fundamentals

Setup Steps

Step 1: Start OpenOCD

```
openocd -f rp2350.cfg
```

Step 2: Launch GDB

```
gdb-multiarch hello.elf
```

Step 3: Connect to target

```
target remote :3333
```

Step 4: Reset + halt

```
monitor reset halt
```

Step 5: Set breakpoint

```
break main
```

Then: continue (c)

What Each Does

openocd

Bridges USB probe
to GDB via TCP

Listens on port 3333

gdb-multiarch

ARM-aware debugger
Loads ELF symbols

target remote

GDB connects to
OpenOCD server

monitor reset halt

Reset chip + stop
at very first instr
Clean starting state

Breakpoints

GDB Breakpoint Types

How They Work

Normal Execution

```
MOV r0, #5
```

```
MOV r1, #3
```

```
BL printf
```

With Breakpoint

```
MOV r0, #5
```

```
MOV r1, #3
```

STOP

```
BL printf
```

paused

CPU halts BEFORE
executing breakpoint
instruction

Now you can inspect

GDB Breakpoints

break main

Stop at function

By symbol name

break *0x10000340

Stop at exact addr

By hex address

info break

List all active
breakpoints

continue (c)

Resume running
until next break

delete 1

Remove breakpoint #1

Stack in Action

Runtime Stack Analysis

Before Call

SP = 0x20082000

(empty)

(empty)

(empty)

(empty)

Stack is clean
SP at top of
SRAM space

Ready to call
main()

After PUSH

PUSH {r4, lr}

(empty)

(empty)

LR saved

r4 saved

SP moved down
by 8 bytes
2 regs x 4 bytes

GDB: info regs
Watch SP change!

Key Points

PUSH saves

Preserves regs
before function
body runs

POP restores

Puts values
back when func
returns

Watch in GDB

x/4xw \$sp
See stack data

stepi

Step 1 instr
watch stack
change live

LDR Instruction

ARM Load Instructions

How LDR Works

Instruction:

```
LDR r0, [pc, #12]
```

Step 1: Calculate addr

```
addr = PC + 12
```

Step 2: Read memory

```
value = *(addr)
```

Step 3: Load into reg

```
r0 = value
```

r0 now holds the
address of our
"hello, world" string

Why It Matters

String Loading

printf needs addr
of string in r0
r0 = first argument

PC-Relative

Address computed
relative to current
PC position
Works from any addr

The Attack Point

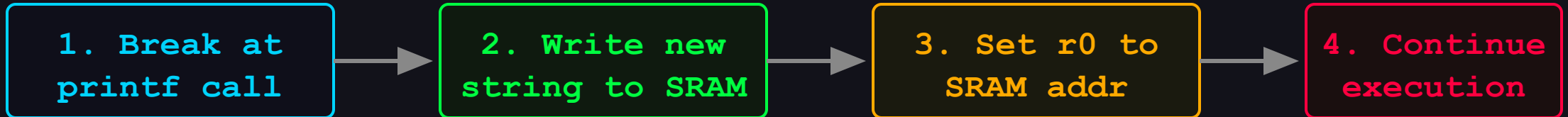
If we change r0
AFTER the LDR
printf prints OUR
string instead!

This is the hack!

The Attack Plan

Exploit Strategy

Attack Flow (4 Steps)



`printf` reads `r0`, prints "hacky, world"!

Normal Flow

```
LDR r0, ="hello"
```

```
BL printf
```

Output:

"hello, world"

Prints original string

Hacked Flow

```
LDR r0, ="hello"
```

```
r0 = 0x20040000
```

```
BL printf
```

Output:

"hacky, world"

Prints our string

Failed vs Real Hack

Attack Methodology

Failed Attempt

The Bad Idea

Set r0 to point
at a string literal
like "hacky"

Why It Fails

r0 only holds a
32-bit number
Not a string itself!

```
set $r0 = "HACK"
```

GDB interprets this
as an address value
pointing to garbage

Result: CRASH

or prints garbage

Real Hack

The Right Way

1. Write string
bytes to SRAM
2. Point r0 to
that SRAM addr

GDB Commands

```
set {char[13]}0x20040000
```

```
= "hacky, world"
```

```
set $r0 = 0x20040000
```

String exists in
writable SRAM
r0 points to it

"hacky, world" printed!

Writing to SRAM

Memory Manipulation

SRAM at 0x20040000

Before (empty)

```
00 00 00 00 00 00 00 00
```

```
00 00 00 00 00 00 00 00
```

After writing

```
68 61 63 6b 79 2c 20 77
```

```
h a c k y , w
```

GDB Command:

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

Verify with:

```
x/s 0x20040000
```

Why SRAM?

SRAM = writable

RAM at 0x20000000

We can write any
data here via GDB

Flash = read-only

XIP at 0x10000000

Cannot write to it
during execution

That's why we use RAM

Choosing Address

0x20040000 is safe
Far from stack
and heap regions

Null terminator

\0 ends the string

Register Hijack

Control Flow Attack

Before Hijack

r0 loaded by LDR:

```
r0 = 0x10001234
```

Points to flash:

```
"hello, world\r\n"
```

printf will read r0
and print that string

The Hijack Command

```
set $r0 = 0x20040000
```

Now r0 points to
OUR string in SRAM
instead of flash

After Hijack

r0 now contains:

```
r0 = 0x20040000
```

Points to SRAM:

```
"hacky, world"
```

Then: continue

printf reads r0
Follows pointer
to 0x20040000
Finds "hacky, world"
Prints it!

Output changed
without touching code

GDB vs Ghidra

Static vs Dynamic Analysis

GDB (Dynamic)

Live analysis

Program is running
on real hardware

Capabilities

Set breakpoints
Read/write memory
Modify registers
Step instructions
Watch values change

Best For

Live modification
Runtime behavior
Testing exploits
Verifying attacks

Needs running target

Ghidra (Static)

Offline analysis

Just the binary file
No hardware needed

Capabilities

Disassembly view
Decompile to C
Find functions
Cross-references
String search

Best For

Planning attacks
Understanding code
Finding targets
Mapping functions

Works with just ELF