Weird CPU

Powered by

SEcube™

Definitions

- **VulnBox**: the VM given to the teams by the organization
- **WeirdCPU proxy**: the service that runs on the VulnBox
- **FPGA service**: the service that manages the FPGA, it's not managed by the teams, teams can only contact it to have their programs executed
- **User**: whoever uses the service. It could be you, a teammate, the Checker or an attacker
- **Checker**: is a legitimate user that performs automated legit operations on the service in order to ensure its health. Its actions include the retrieval of flags and other non flag related operations.
- **RN:RM**: registers from RN to RM (e.g. R1:R23 are the registers from R1 to R23)

Introduction

The challenge is based on a CPU built on top of an FPGA. It's a full fledged A/D challenge, each round a new flag is loaded and attackers need to attack the opponents each round in order to get the new flag.

Below the documentation of the CPU and the communication services can be found.

CPU

The CPU is an 8-bit simplified CPU

Overview

- No RAM, no Program Memory (no Program Counter), instructions are fed directly to the CPU (see "CPU communication")
- A few simple instructions (see "Instructions"), with fixed length (3 bytes)
- 64 8-bit registers
- No Turing completeness
- Stateless between executions
- A special bus called "cu_out" used to output data (it cannot be read by the CPU)

Instructions

Instructions are 3 bytes long, the first byte represents the OPCODE (note: only the 6 rightmost bits of the OPCODE are considered by the CPU), the remaining 2 bytes represent the data of the instruction.

List format: "INSTRUCTION_NAME (OPCODE): DESCRIPTION"

- NOP (0x00): Do nothing (since instructions are fed from an "unknown source" slower than the CPU, a lot of NOP instruction are executed in between of the user instructions)
- WRITE (0x01): Load a fixed value into a register, the first byte of the data represent the destination register, the second byte is the value
- COPY (0x02): Copy the value of a register into another register, the first byte of the data represent the source register, the second byte is the destination register
- NOT (0x04): NOT the value of the register and store it in R0, the first byte of the data represent the source register
- AND (0x05): AND the values of the two registers specified by the bytes of the data and stores the result in R0
- OR (0x06): OR the values of the two registers specified by the bytes of the data and stores the result in R0

- XOR (0x07): XOR the values of the two registers specified by the bytes of the data and stores the result in R0
- DISABLE_RS (0x08): Disable forever (until reset) read and write operations on the registers from R32 to R61, except for CHECK_RS
- CHECK_RS (0x09): Compare registers from R0 to R31 with registers from R32 to R61, if all pairs are equals a 0x1 is written to R0, else a 0x0 is written to R0
- READ (0x20): Output the value of a register into “cu_out”: the first byte of the data represents the source register

If an OPCODE not present in the list above is issued to the CPU it’s treated as a NOP.

Program examples

Simple XOR

0x01 0x01 0xAA	R1 = 0xAA
0x01 0x02 0x33	R2 = 0x33
0x05 0x01 0x02	R0 = R1 AND R2 (result 0x11)

Some instructions

0x01 0x2D 0x35	R45 = 0x35
0x01 0x3C 0x12	R60 = 0x12
0x07 0x2D 0x3C	R0 = R45 XOR R60 (result 0x27)
0x02 0x00 0x18	R24 = R0
0x05 0x18 0x2D	R0 = R24 AND R45 (result 0x25)

Illegal operations

0x08 0x00 0x00	DISABLE R32:R63 (executed correctly)
0x02 0x20 0x00	R0 = R32 - NOT EXECUTED because R32:R63 registers are disabled (by DISABLE_RS)
0x2A 0x01 0x02	UNKNOWN OPCODE, treated as NOP

Challenge example

0x01 0x20 0x12	R32 = 0x12
0x01 0x21 0x34	R33 = 0x34
...	
0x01 0x3F 0x56	R63 = 0x56
0x08 0x00 0x00	DISABLE R32:R63
USER INSTR 1	User Instruction 1
USER INSTR 2	User Instruction 2
...	User Instructions
USER INSTR N	User Instruction N
0x09 0x00 0x00	Check Registers
0x20 0x00 0x00	cu_out = R0

The colored instructions (orange and green) are specific to the challenge (see “Challenge description”).
Orange instructions are prepended to the user program by the FPGA service in order to load the 32 registers (from R32 to R63) and then disable access to them.
Green instructions are appended to the user program by the FPGA service in order to check if the challenge is completed.

Challenge

Challenge description

The challenge is based on an FPGA. The VHDL source code of the IP core of the CPU implemented on the FPGA can be downloaded from the WeirdCPU proxy homepage.

This document contains the correct specification for the CPU, anything present in the VHDL source code not specified in this document should be treated as malicious or faulty code.

The objective of the challenge is to put the values of registers R32:R63 into the registers R0:R31, by specification the CPU is unable to read the R32:R63 registers after the instruction DISABLE_RS, so it should be impossible to achieve it without knowing the correct values.
Do not try to bruteforce the values, it will only end up in a global DoS and your exclusion from the competition.

The challenge receives a program from the user (see “CPU communication”) and executes it on the CPU.

The user program is modified in order to prepare the environment for the challenge, the new program will look like this:

- R32 to R63 are loaded with bytes, known only by the Checker, using the WRITE instruction (32 WRITE instructions), these 32 bytes are already known by the FPGA service they are not present in any request
- A DISABLE_RS instruction is executed (1 DISABLE_RS instruction)
- The original user program is executed (user program)
- A CHECK_RS instruction is executed in order to write the result of the challenge in R0 (1 CHECK_RS instruction)
- A READ instruction is executed in order to get (and check) the result inside R0 (1 READ instruction)

For an example of the resulting program see “Challenge example” in the “Program examples” section.

The legitimate requests to the service are made by the Checker which already knows the correct bytes and it’s going to set the registers R0:R31 using the documented CPU instructions, if these requests (or the related responses) are blocked the team is going to lose SLA points.

Legitimate requests can also be issued (always by the Checker) in order to check some functionality (without doing any operation to retrieve the flag, if you block these requests (or the related responses) you’re going to lose SLA points.

Malicious requests should be aimed to retrieve the flag without knowing the correct bytes for the registers.
It's not mandatory that a request is only aimed to retrieve flags, any kinds of requests can be received.

Service

The service "WeirdCPU" exposed on port 5000 on your VulnBox is responsible for relaying information between the game network and the FPGA.

The CPU contains vulnerabilities, you cannot modify the FPGA behaviour, but you can filter requests on your VulnBox, a simple script is already given (see "WeirdCPU proxy description").

The WeirdCPU proxy only sends the user programs to the FPGA service, new flags do not come through the WeirdCPU proxy (on your VulnBox), they're already on the FPGA service, waiting to be released upon the completion of the challenge.

Upon request from a user the service must send the request (with the program) to the FPGA and wait for the response that has to be relayed to the client (the response contains the flag if the challenge is solved, see "Challenge description").

The response is a JSON typed as follows:

```
{ ok: true; flag: string; }  |  { ok: false; error: string; }
```

You can only get the current round flag, the previous flag is no longer available once the Checker puts the new one.

CPU communication

In order to communicate with the CPU the WeirdCPU proxy on your VulnBox contacts (with a unique token) the FPGA service on another VM not managed by you.

The input for the CPU is a program. A program is a sequence of instructions (3 bytes each). The program maximum length is defined in the "Constraints" section, the value refers to the user program, not the modified one (as described in "Challenge description").

The WeirdCPU proxy receives the program through an HTTP request, it analyzes it with a WAF (the WAF is described in the "WeirdCPU description" section) and it sends it to the FPGA service.

Once a program arrives to the FPGA service it's enqueued waiting for its turn on the FPGA.
The queue has a global limit (defined in the "Constraints" section), if it's exceeded a specific error is returned as response.

Note: if you fill the queue from the WeirdCPU proxy of another team you're going to cause a DoS for everyone (including you), and, BTW, this behaviour is against the rules.

The program is modified in order to prepare the challenge environment (see "Challenge description").

Once its turn comes the FPGA is resetted (see the "reset" signal) and the program is executed.

The last 2 instructions of the program are CHECK_RS, which writes the result of the challenge in R0, and READ 0x00, which outputs R0 to "cu_out" that is read by the FPGA service. If "cu_out" is 0x01 then a flag is sent as response for the request; if "cu_out" is not 0x01 an error is returned as response.

The FPGA service responds to the WeirdCPU request, and the WeirdCPU proxy should relay the response to the user.

An example of the complete program that is going to run on the FPGA can be found in the "Program examples" section.

WeirdCPU proxy description

The service WeirdCPU proxy, as described before, is responsible for connecting the user to the FPGA.
The WeirdCPU proxy provides a WAF (Web Application Firewall) that should be used to block (or edit) malicious requests.
The WeirdCPU proxy identifies itself on the FPGA service with a token (FPGA_TEAM_TOKEN from the environment variables inside the container), you should not change it.

Inside the waf.py file a custom filter can be written. A program can be modified or its execution can be prevented.

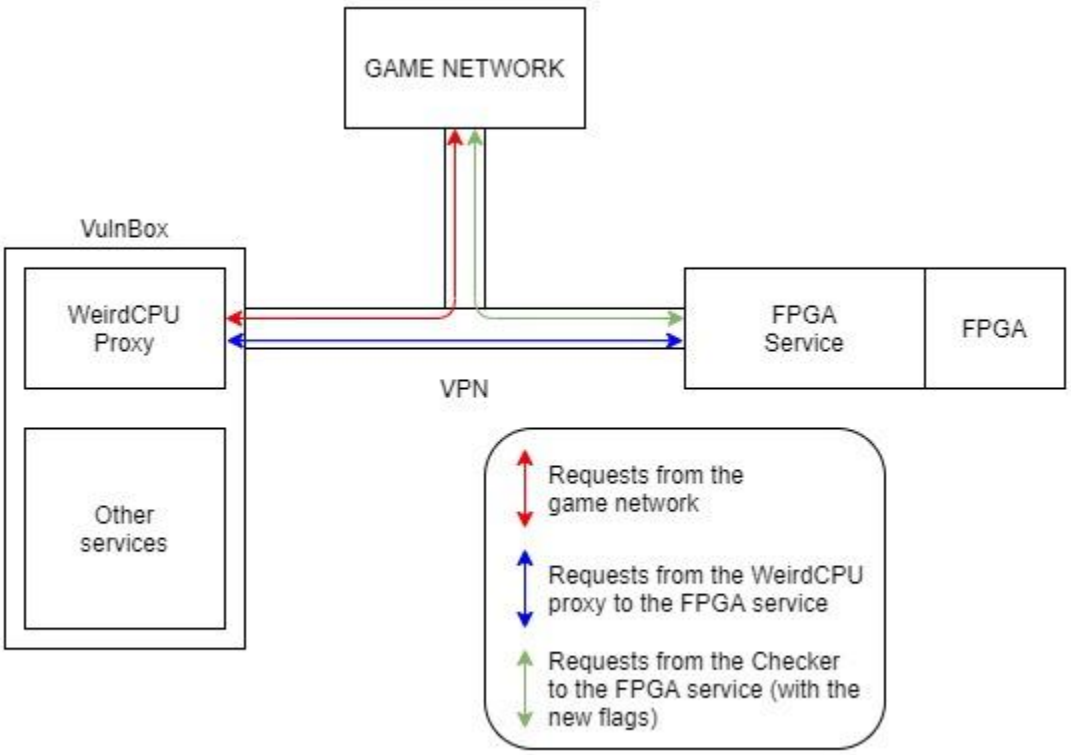
The filter_program function is called each time a user sends a program, the function should return the program to be sent to the FPGA service (original or modified) or None (in order to prevent the execution).
The filter_program function can also be used to get useful logs from the programs.

waf.py contains some examples.

Remember to rebuild the container after modifying waf.py (or any other source file).

Network diagram

Requests from the game network (red) are made by you, your team, the Checker and the other players (attackers).
Requests from the Checker (geen) are made by the checker in order to load the new flags.



Constraints

- The program length should be a multiple of 3
- Max program length: 1024 bytes
- FPGA service queue limit: 30000 requests
- Max execution time (should never be reached): 1s

Notes

DoS attacks (like making a lot of requests to a service) are prohibited.