

2025

# CTF INTERNACIONAL METARED ETAPA MEXICO

WRITEUP

## CHALLENGE

BitWise Bias

## Description

A privacy-focused startup called BitSecure launched a new mathematical proof-of-work system to protect sensitive communication metadata from fraudsters. Their backend relies on complex bitwise puzzles using only a small set of allowed variables to prevent reverse engineering. Unfortunately, due to weak verification logic and predictable randomness, a skilled attacker might bypass the system and extract the protected environment variable... possibly revealing customer data..

**File:** server.py

# Solution:

## 1. What's Provided

- **server.py**: Service that:
  - Uses `z3` with `BitVec(32)` and operators `~ & ^ |`.
  - Limits variables to `i, I, l` (three visually similar characters to cause parsing mistakes).
  - Generates random infix expressions and asks for their **RPN equivalent**.
  - Checks semantic equivalence with `Z3`; if not equivalent, it can print a **counterexample model** to help debugging.

## 2. Challenge Analysis

Key points after reading `server.py`:

- **32-bit domain**: Everything is evaluated as a `BitVec(32)`. `~` is a **bitwise NOT**, not a logical NOT.
- **Operators and precedence**:
  - Highest  $\rightarrow$  lowest: `~` (unary), `&`, `^`, `|`
  - Associativity: `~` **right-associative** (unary prefix), `&`, `^`, `|` **left-associative**.
- **Required format: RPN** (postfix). Examples:
  - `i & ~l  $\rightarrow$  i l ~ &`
  - `~(i | I) ^ l  $\rightarrow$  i I | ~ l ^`
- **Validation with Z3**: The server builds both ASTs (yours from RPN and its own from infix) and checks *equivalence*. On failure, it may print a model showing variable assignments where they differ.

## 3. Solution Methodology

1. **Connect to the service** (localhost:4443 via Docker or remote).
2. **Extract the expression** from the line (strip surrounding text); only keep `[iIl()~&^|]` and spaces.
3. **Convert infix  $\rightarrow$  RPN** using a variant of the **Shunting-Yard algorithm**:
  - Support **unary ~**: treat it as a 1-argument operator, **highest precedence, right-associative**.
  - When outputting an operand, **flush** any pending unary `~` operators so they appear **after** the operand.
  - Handle parentheses properly.
4. **Send the RPN** to the server.
5. **Repeat** until the server prints the flag.

The flag: `flagmx{bitw1se_l3ak4ge}`

#### 4. Solution Script:

```
from pwn import *
import re

context.log_level = 'info'

HOST = 'localhost'
PORT = 4443

# Converts a bitwise expression from infix notation (normal math format)
# into Reverse Polish Notation (RPN), which is required as input to the
# challenge.
def to_rpn(expr):
    # Operator precedence for bitwise operations
    precedence = {'~': 3, '&': 2, '^': 1, '|': 0}
    output = []
    stack = []

    # Tokenize the expression: variables, numbers, operators, parentheses
    tokens = re.findall(r'[\~&^|()]\d+|[iIl]+', expr)

    for token in tokens:
        if re.fullmatch(r'\d+[iIl]+', token):
            # Operand (number or variable)
            output.append(token)
        elif token == '(':
            stack.append(token)
        elif token == ')':
            # Pop from stack until opening parenthesis
            while stack and stack[-1] != '(':
                output.append(stack.pop())
            stack.pop() # Discard '('
        elif token in precedence:
            # Pop operators with higher or equal precedence
            while stack and stack[-1] != '(' and precedence.get(stack[-1], -1) >= precedence[token]:
                output.append(stack.pop())
            stack.append(token)

    # Empty any remaining operators from the stack
    while stack:
        output.append(stack.pop())

    return ' '.join(output)

# Connects to the challenge server, receives infix expressions,
# converts them to RPN using `to_rpn()`, sends the answer, and repeats
# until either failure or flag is received.
def main():
```

```

io = remote(HOST, PORT)

while True:
    # Read a line from the server
    line = io.recvline(timeout=5).decode().strip()
    if not line:
        break

    log.info(f'SERVER: {line}')

    # Detect challenge line containing a bitwise expression
    if re.match(r'^[()~&|^0-9iIl\s]+$', line):
        try:
            # Convert to RPN and send it
            rpn = to_rpn(line)
            log.success(f'RPN: {rpn}')
            io.sendline(rpn)
        except Exception as e:
            log.error(f'Conversion error: {e}')
            break

        if 'flagmx' in line:
            log.success(f'FLAG: {line}')
            break

    io.close()

# Run the exploit
if __name__ == '__main__':
    main()

```

### 1) Robust expression extraction

Uses a permissive regex to capture the **longest substring** containing only `iIl()~&|^|` and spaces. This tolerates noisy prompts.

```

python
CopiarEditor
m = re.findall(r"[iIl\~\&\^\|\|\(\)\s]+", line)
candidate = max(m, key=len).strip()

```

### 2) Infix→RPN conversion with unary ~

- Tokenize: classify into `var`, `(`, `)`, binary op (`&` `^` `|`) and `uop` for unary `~` (if at start, after another operator, or after `(`).
- Precedence: `u~:3`, `&:2`, `^:1`, `|:0`.
- Associativity: `u~: right`, binary ops: `left`.
- Practical rule: after emitting an **operand** or closing `)`, **pop** all pending `u~` so they're in postfix position.

Examples:

- Input:  $i \ \& \ \sim 1 \rightarrow$  Tokens:  $i, \&, u\sim, 1 \rightarrow$  **RPN**:  $i \ 1 \ \sim \ \&$
- Input:  $\sim(i|I) \ ^ \ 1 \rightarrow$  **RPN**:  $i \ I \ | \ \sim \ 1 \ ^$
- Input:  $i \ ^ \ I \ ^ \ 1$  (left-associative)  $\rightarrow$  **RPN**:  $i \ I \ ^ \ 1 \ ^$

### 3) Send & loop until the flag

The script reads lines; when it detects an expression, it computes RPN and sends it. If the server sends back the flag (`flagmx{...}`), it stops.

### Running Locally

```
bash
CopiarEditar
# Build & run the challenge
docker build -t bitwise-bias .
docker run -d -p 4443:4443 --name bitwise-bias bitwise-bias

# In another terminal: (or use remote HOST/PORT)
python3 solve.py
```

### Conversion Examples (Sanity Checks)

- $i \ \& \ I \ | \ 1 \rightarrow i \ I \ \& \ 1 \ |$
- $i \ | \ I \ \& \ 1 \rightarrow i \ I \ 1 \ \& \ |$  (because  $\& > |$ )
- $\sim i \ \& \ (I \ ^ \ 1) \rightarrow i \ \sim \ I \ 1 \ ^ \ \&$
- $i \ ^ \ \sim I \ | \ \sim(1) \rightarrow i \ I \ \sim \ ^ \ 1 \ \sim \ |$

### Common Mistakes and How to Avoid Them

- **Treating  $\sim$  as binary**: always detect **unary** in parsing.
- **Forgetting associativity**:  $\wedge$ ,  $\&$ , and  $|$  are **left-associative**;  $\sim$  (unary) is **right-associative**.
- **Not flushing  $\sim$  after operands**: in RPN,  $\sim$  appears **after** the operand it applies to.
- **Mismatched parentheses**: always validate and raise errors if unbalanced.
- **Confusing variables**  $i$ ,  $I$ ,  $1$ : they are **different symbols** (intentionally similar visually).

## 5. Lessons Learned

- **Robust parsing** and **Shunting-Yard** with unary operators.
- **RPN** as a canonical form for evaluating/transforming expressions.
- **BitVec in Z3**: understanding how **NOT** ( $\sim$ ), **AND** ( $\&$ ), **XOR** ( $\wedge$ ), and **OR** ( $|$ ) work on 32-bit values.
- Using **counterexample models** from Z3 to debug equivalences.
- Automating challenge solving with **pwntools** and handling noisy prompts.

## 6. Conclusions

- **Understanding operator precedence** is critical in both programming and binary exploitation, where misinterpretation of expression order can lead to logical errors or exploitable vulnerabilities.
- **Parsing with precision** (handling unary operators correctly) prevents subtle but critical bugs.
- This challenge reinforces the importance of **clear algorithm design** (Shunting-Yard) and **defensive coding** when dealing with unpredictable input.
- Automated solvers reduce human error in repetitive parsing and transformation tasks.

## 7. Digital Privacy Implications

While this challenge is framed as a parsing exercise, its mechanics reflect **real-world privacy and security risks**:

- Incorrect parsing or operator precedence bugs in **encryption/decryption logic** can weaken cryptographic protections, leading to **data leaks**.
- Flaws in bitwise logic are particularly relevant in **access control systems, masking sensitive data, or flag-based security checks**—where a single precedence error might bypass security measures.
- The ability to **reverse-engineer and replicate logic exactly** is essential in auditing privacy-critical software; a failure to do so could mean attackers exploit misinterpretations to exfiltrate personal information.
- In digital privacy contexts, **input sanitization and parser correctness** are just as important as cryptographic strength—errors at this layer can make otherwise secure systems vulnerable.