

CPU Vulnerabilities

Finding Bugs Where Software Ends

Michael Schwarz | FOSAD Summer School 2026

Lab #2:

Leak a Cache Line



Lab #2: LoongLeak in QEMU

Hands-on: Find the hidden line



We provide a QEMU that runs a LoongArch CPU affected by LoongLeak. The victim writes one secret cache line. Your code has to get it back.



The Setup

The files

- `qemu-loongarch64`: Emulator for LoongLeak
- `challenge/`: attack template and starter code
- `la-gcc.py`: compiler helper

The rules

- Write LoongArch user-space code (in C)
- Exploit LoongLeak
- Recover the secret from the cache



Start Simple

```
wget https://fosad.cpu.wtf/lab2/bundle.zip  
unzip bundle.zip  
cd challenge  
./run_challenge.sh
```

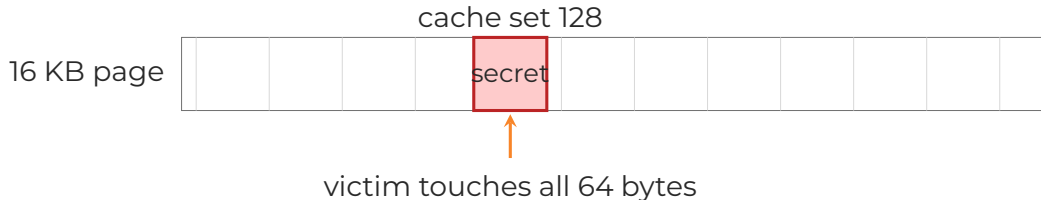
- The starter should print zeros and FAIL.

Takeaway

A FAIL here is good news: your function ran.



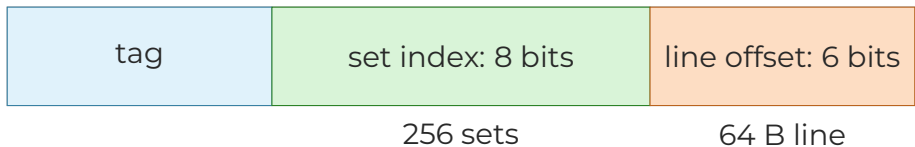
What the Victim Does



- Victim writes a 64-byte secret to cache set 128
- Other lines contain readable decoy text
- Then your function is called



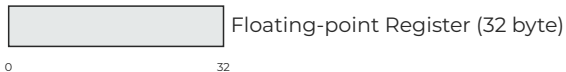
Find the Right Cache Set



For cache set 128, the set-index bits must be 128



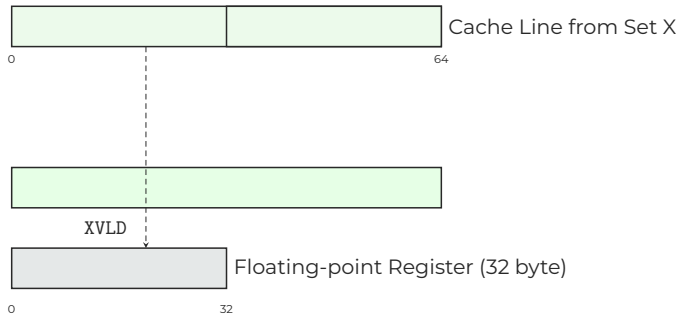
Why Do We Prime?



- XVLD “fixes” the target set
- FLD.S loads 8 bytes from an address, rest from the set “fixed” by XVLD



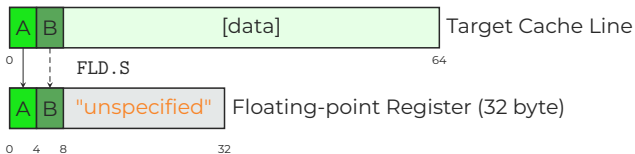
Why Do We Prime?



- XVLD “fixes” the target set
- FLD.S loads 8 bytes from an address, rest from the set “fixed” by XVLD



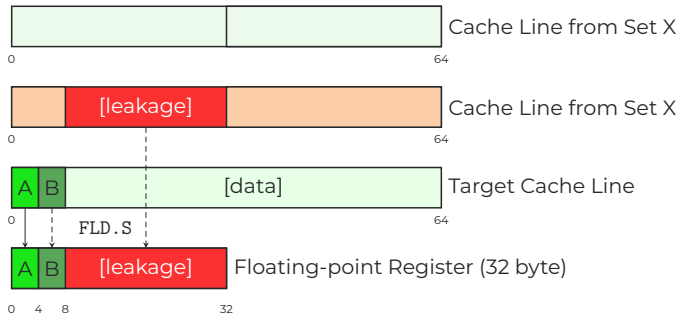
Why Do We Prime?



- XVLD “fixes” the target set
- FLD.S loads 8 bytes from an address, rest from the set “fixed” by XVLD



Why Do We Prime?



- XVLD “fixes” the target set
- FLD.S loads 8 bytes from an address, rest from the set “fixed” by XVLD



Read the Two Macros

```
#define LOONGLEAK_PRIME(address) \
__asm__ volatile(  
    "xvld $xr16, %0, 0\\n\\t"  
    "dbar 0\\n\\t" ... )
```

```
#define LOONGLEAK_READ(address, buffer,  
    offset) \
__asm__ volatile(  
    "dbar 0\\n\\t"  
    "fld.s $f0, %0, %c2\\n\\t"  
    "xvst $xr0, %1, 0\\n\\t" ... )
```

Hands-on: Your job

Fill in `student_leak` in `challenge/leaker.c`. No assembly needed.



Run It: What Should You See?

```
cd challenge  
./run_challenge.sh
```

- The victim writes one 64-byte line, then calls `student_leak`.
- You only need this one line.
- The text dump helps you debug. `PASS` checks every byte.

Takeaway

The program tells you: recovered line, then `PASS` or `FAIL`.



Three Ways to Fail

No reads yet

Recovered output line (text):

.....
.....

FAIL: secret not recovered

Wrong set: 127, not 128

Recovered output line (text):

victim-cache-line-readable-data-
victim-cache-line-readable-data-

FAIL: secret not recovered

Almost there... but broken

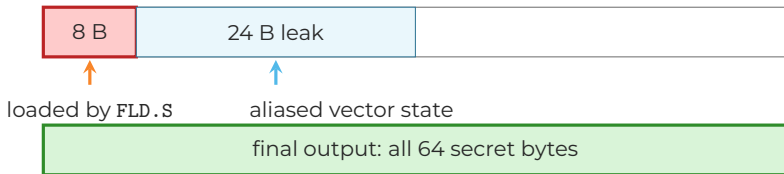
Recovered output line (text):

.....56789abcdef0123456789abc.....56789abcdef012345678901

FAIL: secret not recovered



Why Is “Almost Right” Still Wrong?



- A 32-byte store contains 8 bytes defined by FLD.S
- Other 24 bytes are the leaked values
- A second read is required for the 8 bytes that were loaded in the first read
- Merge them into the correct positions



Your Turn: Write the Leaker

```
cd challenge  
./run_challenge.sh
```

1. Calculate the prime address from `target_set`
2. Prime before each read: offsets 0, 40, 32, 24
3. Store each 32-byte result in a temporary buffer
4. Join the pairs in the output buffer

```
Recovered output line (text): LL{0123456789abcdef0123456789abcdef  
                                0123456789abcdef012345678901}  
PASS: secret recovered from stale floating-point bytes
```



Bonus: Change the Victim

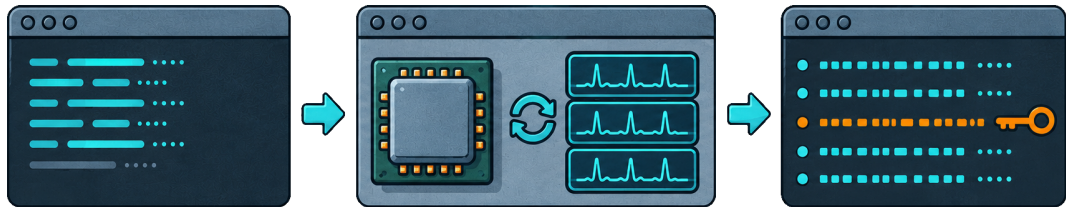
Hands-on: Bonus

Give the victim a new secret. Run the challenge again.

- The new pattern should appear without changing QEMU
- So QEMU is not hard-coded to one magic pattern
- Keep it fake, small, and repeatable



Session 2 Close



**We made a real existing CPU leak into actual data -
deterministically, repeatable.
Unprivileged software in, secret line out.**