[H-1] the Token contract fails to check remaining balance of target address when call burn function to burn tokens, which will result in underflow risk and the attacker would get large amount of tokens from nowhere.

Description: In contract `token`, it calls burn(address account, uint256 value) to reduce remaining balance of `account` and reduce `totalsupply` and it is realized in assembly code.

Crucially, before reducing the token holder's balance, the assembly code fails to check if the `account` possesses a balance greater or equal to the `value` being burned.

The relevant vulnerable code within the assembly code is in `ERC20Internals.sol`:

```
    let accountBalanceSlot := keccak256(ptr, 0x40)
    let accountBalance := sload(accountBalanceSlot)
    sstore(accountBalanceSlot, sub(accountBalance, value))
```

Since the EVM `sub` operates on unsigned 256-bit integers and the balance check is missing, performing the substraction is negtive(i.e., `accountBalance < value>`) causes the value to underflow(wrap around 2^{256}), leading to a massive balance.

Impact: This is a critical vulnerability that allows for unauthorized token minting(creation).

- Unauthorized token creation: An attacker, even with zero balance, can call public `burn` function to mint massive tokens out of no where,
- Asset Inflation: The resulting underflow will set the attacker's balance to a number close to 2^{256}, effectively creating tokens out of thin air and granting the attacker immense wealth in the protocol.
- Protocol Failure: The totalsupply is also incorrectly reduced, leading to inconsistences and breaking fundamental invariants of the ERC-20 standard, making the token unusable.

Proof of Concept: The following Foundry Test demonstrates how an attacker with zero balance can mint massive amount of tokens out of the air by calling the public `burn` function by performing the underflow attack. We assert that the resulting balance is bigger than the initial balance and matches the precise mathematical underflow value.

Assuming token.balanceOf(account) initially returns 0 and the attacker attempts to burn 51×10^{18} tokens.

```
function test_burnRevert2() public {
        address account = makeAddr("account");
        uint256 initialBalance = token.balanceOf(account);

        token.burn(account, 51e18);
        uint256 balanceAccount = token.balanceOf(account);
        console.log("balance of account:", balanceAccount);
        assertTrue(balanceAccount > initialBalance,"Balance must increase due to
underflow.");
        uint256 expectedUnderflowValue = type(uint256).max-(51e18-initialBalance)+1;
        assertEq(expectedUnderflowValue,balanceAccount);
    }
```

Recommended Mitigation: A balance checking must be implemented in assembly code before the substraction operation. If the balance is insufficient, the transaction must be reverted.
This requires adding an explicit check using the `lt`(less than)opcode and the `revert` operation.

Suggested Fix(within `_burn` assembly):

```
    let accountBalanceSlot := keccak256(ptr, 0x40)
    let accountBalance := sload(accountBalanceSlot)
+   if lt(accountBalance, value){
+       mstore(0x00, shl(224, 0xe450d38c))
+       mstore(add(0x00, 4), account)
+       mstore(add(0x00, 0x24), accountBalance)
+       mstore(add(0x00, 0x44), value)
+       revert(0, 0x64)
+    }
+
    sstore(accountBalanceSlot, sub(accountBalance, value))
```