cją i chce odzyskać swoją część etheru przed jej realizacją, może przedstawić i przyjąć propozycję specjalnego typu, dotyczącą utworzenia nowej DAO. Posiadacze tokenów, którzy poparli propozycję, mogą wówczas podzielić DAO i przesunąć swoje środki w etherze do nowej organizacji, doprowadzając do tego, że reszta będzie mogła dysponować jedynie swoim etherem.

Niestety, sposób, w jaki wdrożono narzędzie podziału, przyczynił się do powstania katastrofalnego błędu wielobieżności, który przesądził o podatności The DAO na atak[9]. Innymi słowy, ktoś mógł rekurencyjnie dzielić The DAO i wypłacać w nieskończoność środki w ETH równe jego początkowej inwestycji jeszcze przed zarejestrowaniem wypłaty w umowie pierwotnej The DAO.

Oto słaby punkt znaleziony w pliku umowy Solidity pod nazwą DAO.sol:

```
function splitDAO(
  uint _proposalID,
  address _newCurator
) noEther onlyTokenholders returns (bool _success) {

  ...
  // [Added for explanation] The first step moves Ether and
      assigns new tokens
  uint fundsToBeMoved =
       (balances[msg.sender] * p.splitData[0].splitBalance) /
       p.splitData[0].totalSupply;
  if (p.splitData[0].newDAO.createTokenProxy.value(fundsToBeMo-
ved)(msg.sender) == false) //
[Added for explanation] This is the line that splits the DAO be-
fore updating the funds in
the account calling for the split

  ...
  // Burn DAO Tokens
  Transfer(msg.sender, 0, balances[msg.sender]);
  withdrawRewardFor(msg.sender); // be nice, and get his rewards
  // [Added for explanation] The previous line is key in that it
is called before
totalSupply and balances[msg.sender] are updated to reflect the
new balances after the split
has been performed
```

[9] Wielobieżność jest cechą oprogramowania, którego procedura może zostać przerwana podczas wykonywania, a następnie uruchomiona od początku, gdy jednocześnie pozostała część procedury oczekuje w kolejce do wykonania.

```
  totalSupply -= balances[msg.sender]; // [Added for explanation]
This happens after the
split
  balances[msg.sender] = 0; // [Added for explanation] This also
happens after the split
  paidOut[msg.sender] = 0;
  return true;
}
```

Jak widać, The DAO odwołuje się do tablicy balances w celu określenia, ile tokenów DAO może zostać przesuniętych. Wartość p.splitData[0] jest własnością propozycji przedkładanej w The DAO, a nie własnością samej The DAO. Właśnie to, w połączeniu w faktem, że WithdrawRewardFor jest wywoływane, zanim dochodzi do aktualizacji balances[], sprawiło, że atakujący mógł wywoływać fundsToBeMoved w nieskończoność, jako że jego saldo ciągle zwracało pierwotną wartość.

Bliższe przyjrzenie się withdrawRewardFor() ujawnia warunki, które uczyniły to możliwym:

```
function withdrawRewardFor(address _account) noEther internal re-
turns (bool _success) {
  if ((balanceOf(_account) * rewardAccount.accumulatedInput()) /
totalSupply < paidOut[_account])
    throw;

  uint reward =
    (balanceOf(_account) * rewardAccount.accumulatedInput()) /
totalSupply - paidOut[_account];
  if (!rewardAccount.payOut(_account, reward)) // [Added for
explanation] this is the statement that is vulnerable to the re-
cursion attack. We must go deeper.

    throw;
  paidOut[_account] += reward;
  return true;
}
```

Przyjmując, że pierwsza instrukcja przyjmuje wartość „fałsz", zostaje uruchomiona instrukcja oznaczona jako słaby punkt. Musimy wykonać jeszcze jeden krok, by w pełni zrozumieć, jak haker był w stanie to wykorzystać. Gdy po raz pierwszy wywoływane jest withdrawRewardFor (a więc gdy haker ma jeszcze prawdziwe środki na koncie, które może wypłacić), pierwsza instrukcja prawidłowo przyjmuje wartość „fałsz", co uruchamia następujący kod: