# 以太坊智能合約安全之研究

林詠章[1]、林久弘[2]
[1, 2]國立中興大學資訊管理學系
[1] iclin@dragon.nchu.edu.tw、[2] a073927339a@gmail.com

## 摘要

區塊鏈技術是以點對點網路為基礎，將資料分散於網路中的每個節點，同時也不需要任何第三方的控管與維護，諸多的特性帶起了虛擬貨幣「比特幣」的發展，成為了全球幣值最高的貨幣，後續也帶起了以智能合約為特點的以太坊平台，其特殊的貨幣「以太幣」成為市值僅次於比特幣的虛擬貨幣，由於智能合約的廣泛應用，使以太坊的使用者逐漸增加，然而在撰寫智能合約中，程式語言「Solidity」因為其特殊的規則與語法，導致眾多已部署的合約都含有許多漏洞及陷阱，這也成為了攻擊者的攻擊目標，如著名的「The DAO 事件」，因此，本論文整理了現今智能合約中常見的漏洞及陷阱，以合約模擬過程並提出解決或避免方式，讓撰寫合約者能有效地避免合約遭受攻擊。

**關鍵詞：區塊鏈、以太坊、智能合約、Solidity**

# The study on Ethereum smart contract security

Iuon-Chang Lin[1], Chiu-Hung Lin[2]
[1, 2] Department of Management Information Systems, National Chung Hsing University
[1] iclin@dragon.nchu.edu.tw, [2] a073927339a@gmail.com

## Abstract

Blockchain technology is based on peer-to-peer network. It distributes data to every node in the network, and does not require any third-party control and maintenance. These features make the virtual currency “Bitcoin” popular which become the currency with the highest currency value in the world. It has also make the Ethereum platform featuring smart contracts popular too. It’s special currency “Ether” becomes the virtual currency with the market value that is second only to the bitcoin. Due to the various application of smart contracts, the users of Ethereum has gradually increased. However, in writing smart contracts, the programming language “Solidity” has many loopholes and traps due to its special rules and grammar, so it has become the target of attackers, such as the famous “The DAO”. Therefore, this paper survey the vulnerabilities and pitfalls in today’s smart contracts, and use the contract simulation process and propose solutions or avoidance methods to make the programmer avoid contract damage

effectively.

**Keywords: Blockchain, Ethereum, smart contract, Solidity**

## 壹、前言

「虛擬貨幣」，又稱為加密貨幣或數位貨幣[9]，而這個詞彙，相信在二到三年前，或許大家都不太熟悉、甚至是從未耳聞，而當「區塊鏈」技術漸漸浮出檯面，連帶著虛擬貨幣等名詞也越來越廣為人知，尤其是 2017/05/12 的知名「WannaCry」大規模電腦綁架事件[10][11]，駭客要求受害者以支付虛擬貨幣「Bitcoin」作為解鎖被綁架檔案的唯一方法，當時比特幣瞬間水漲船高，從大約 1：350 U.S.D，短短幾個禮拜漲到了約 1300 美元，成長率大約為可怕的 400%，拖了此事件的福，各大媒體或電視台開始播報關於虛擬貨幣與區塊鏈的相關新聞，讓更多民眾了解到虛擬貨幣的重要性。

近年來區塊鏈技術的蓬勃發展，也帶動了虛擬加密貨幣的價值提升及提升關注度，甚至已經是許多人做投資(如外幣、股票)的選項之一，而區塊鏈技術以不可竄改性、公開透明、無須信任第三方的等特性聞名，因此非常適合用於金流及貨幣的交易上，比特幣是由化名中本聰在全世界第一個區塊鏈的應用，也是能讓區塊鏈技術發光發熱的代表之一，之後也有許多不同的虛擬加密貨幣發行，如門羅幣、以太幣、音樂幣等等，而目前市面上區塊鏈技術最熱門的兩大應用，分別是單純金流交易的比特幣交易系統[12]，以及導入智能合約以及也擁有其特殊加密貨幣的「以太坊」平台， 以太坊自身也擁有他們專屬的虛擬加密貨幣，稱為「以太幣」，截至 2015 年以來，其資本總額也已經高達 60 億美元，以目前市面上的區塊鏈應用中[13][14]，多數皆是圍繞在不同的貨幣上，也因為有了價值的轉移與流動，不論是區塊鏈底層或是貨幣交易所，都成為了惡意攻擊者的首要目標，尤其是幣值較高的比特幣和以太幣最為常見。

區塊鏈技術本身在資安上雖然安全，但攻擊者仍會從其他方面對貨幣做攻擊，本文要介紹的以太坊，除了擁有自己發行的貨幣之外，還有一個特別的功能，叫做「智能合約」[5]，智能合約是由使用者經由其特殊的計算機代碼，如 Serpent(python-like)、Solidity(javascript-like) 而 Solidity 為目前使用者較為主要使用的選項，在編寫邏輯與功能完成後，合約將部屬進區塊鏈中並等待驗證，驗證完成後即執行合約中的功能，形成一個公開透明，且不能竄改的契約，也因為合約能開發出許多不同的應用如投票、募款等等[1]，相較於區塊鏈 1.0 擁有更多樣性的交易模式，因此以太坊被稱之為區塊鏈 2.0。然而，合約的功能與內容依舊以貨幣的來往或交易為基礎，再加上以太坊是以區塊鏈為基礎的平台，部屬到區塊鏈中的合約除了公開透明，當然也不可竄改，這更開啟了漏洞的大門，一旦智能合約中的原始碼具有漏洞或異常，攻擊者可以透過撰寫攻擊合約並與漏洞合約做互動，達到竊取以太幣，或永久停止合約運轉等情形，在 2018 年的一篇論

文中，研究人員使用其開發的分析工具分析了約 97 萬個現有智能合約，發現仍有高達 34200 個合約具有安全漏洞，可能導致駭客竊取以太幣，甚至刪除合約。

## 貳、文獻探討

### 2.1 以太坊平台介紹

以太坊是一個以區塊鏈技術為底層，並擁有智能合約及自有貨幣以太幣的公共平台[15]，由程式設計師 Vitalik Buterin 於 2013 至 2014 年間，受到比特幣的啟發後提出的概念，在 2014 年透過網路的募資後開始發展，現在由「以太坊基金會」來負責管理，以太坊基金會是一家在瑞士的非營利組織，該基金會的目的是管理通過以太幣預售募集到的資金，從而更好的為以太坊和去中心化的技術生態系統服務。

相對於比特幣，以太坊就是一個能運行智能合約的區塊鏈，並能允許在平台上創建各種應用。簡單來說，區塊鏈(數據結構)+ 智能合約(算法)= 以太坊[6]，相對於比特幣的區塊鏈而言，優勢就是在於它可以容易的實現任何類型的智能合約，由於以太坊的開發歷史較比特幣晚，所以改善了比特幣的某些缺點，例如比特幣的區塊確認時間約為 10 分鐘，以太坊則需約 15 秒左右便可完成。

### 2.2 以太幣

以太幣(Ether)是以太坊平台上的貨幣名稱，生產及運作模式與比特幣十分雷同，目前皆是採用 PoW 共識機制來決定寫入區塊的節點，別給予該節點一定數量的貨幣作為獎勵，當然，以太幣也能在交易所與其他貨幣進行交易，市值是僅次於比特幣的加密貨幣。作為區塊鏈 2.0 的代表，以太幣的價值曾在 2017 年間增長了 230%，增長幅度甚至高過於比特幣。

以太幣同時也是以太坊平台上用來支付交易手續費及訊息處理的媒介，所以當使用者想使用以太坊平台上的服務，首先須創建以太坊錢包，接著必須擁有一定數量的以太幣，取得方式可從參與寫入區塊鏈工作、與其他節點交易、或從直接從指定管道購買，節點可從以太坊官方錢包或其他單位開發的錢包軟體查看自己擁有的以太幣數量。

### 2.3 Gas

在以太坊平台中，當使用者有任何需要將資料寫入區塊鏈的動作，如轉帳、部屬合約、與合約互動等行為，就需要網路上的礦工們來將這些資訊透過 Ethereum Virtual Machine(EVM)[7][8]執行一串指令碼來寫入區塊並認證，同時也會使用 EVM 來維護區

塊鏈的歷史資料,這些需要礦工們幫忙的行為,就需要付出一些手續費 Gas 來獎勵礦工,步驟如下。

1. 使用者部屬合約或發起其他類型交易。
2. EVM 判斷合約中的每行程式碼功能所需的費用，單位為 Gas。
3. 使用者可調整單位 Gas 的金額多寡。
4. 選定手續費後送出等待礦工確認交易。

從以上步驟可以看出，Gas 的多寡並不是使用者自己可選取的，使用者在以太坊上執行的任何功能都有對應的程式碼，EVM 在運算每行程式碼時都會根據不同功能產生對應的 Cost，例如在以太坊上做帳戶與帳戶的轉帳需消耗 21,000 Gas，但是使用者能選擇單位 Gas 的價值。而除了 Gas 的價值可供使用者調整外，另一個被稱之為 Gas Limit。

- Gas Price

  使用者願意支付的單位 Gas 價值，使用者願意提供的金額越高，礦工的收益也越高，通常交易也越快。

- Gas Limit

  使用者願意花的最大值 Gas 數量去完成該筆交易，也可讓系統自動調整。

交易的的手續費計算方式為：

交易手續費 = 使用的 Gas (Gas used) * 選擇的 Gas 價值(Gas Price)

此例子透過 EVM 計算後消耗了 67,849 個 Gas，當使用者的 Gas Limit 選擇大於或小於此數值，分別會發生以下兩種情況：

- Gas Used < Gas Limit

  代表該交易並不需要 Gas Limit 這麼多的 Gas，以太坊會自動將剩餘的 Gas 退還給使用者，並不會造成使用者多扣手續費的情況。

- Gas Used > Gas Limit

  使用者提供的 Gas Limit 還不足以完成該筆交易，那麼該交易會回復初始狀態不會執行，但是使用者仍須付出提供的手續費。

而交易的 Gas Limit 並不是設定越高越好，因為每個 Block 也有其 Gas Limit，以防止交易運算會無上限的消耗系統資源。

### 2.3 智能合約

智能合約(Smart Contract)，是由跨領域的法律學者 Nick Szabo 所提出。他在自己發表的文章中提到智能合約的理念，它的定義如下：『一個智能合約是一套以數字形式定義的承諾,包括合約參與方可以在上面執行這些承諾的協議。』在區塊鏈技術出現以前，智能合約並沒有一個可信任的執行環境，所以一直沒有流行起來。但如今區塊鏈為智能合約提供一個可信的執行環境，理所當然的智能合約在區塊鏈的領域就逐漸嶄露頭角，並且被應用到實際環境中。

在區塊鏈以太坊中的智能合約[4]，具有去中心化、公開透明等特性，其撰寫語言為「Solidity」、「Serpent」等等，而目前 Solidity 為較多人使用的選項，同時也是以太坊官方推薦的程式語言，其架構類似於 JavaScript。

當使用者要在以太坊中部屬合約，大致需要以下步驟:

1. 撰寫智能合約
   使用者可透過自己的需求、撰寫相對應的功能與函式，以語言 Solidity 為例，程式碼會先經由「Solc 編譯器」(Solidity commandline compiler)編譯成二進制的 Contract ByteCode 並轉交給 EVM 執行，EVM 會估算使用者所需 Gas，使用者制定 Gas Price 與輸入錢包密碼即可將合約送出等待驗證。
2. 部屬至區塊鏈網路並驗證
   網路中每個節點皆會收到該合約，並依共識演算法進行寫入區塊鏈的動作，當驗證完畢，合約將被記錄至區塊鏈上。
3. 觸發合約執行
   合約的觸發條件，依照使用者的合約需求及撰寫來定義，如募款合約，當接收到他人募款時即觸發合約內募款功能，當執行合約功能，須回到第二步驟等待驗證,驗證完成後即交易成功。對於觸發合約功能函式可以想像成是一筆交易，只不過交易的對象是合約的 address 而不是帳戶 address，交易類型是合約函式而不是虛擬貨幣。

由於以太坊是以區塊鏈為基礎的平台，所以智能合約的特性與運作，同樣繼承了區塊鏈技術的特性，如合約部署後受到不可竄改性的影響，即使有錯誤或漏洞也無法對合約程式碼做更改。

## 參、方法

在本節中，我們將針對智能合約撰寫時容易忽略的漏洞或陷阱進行整理這些漏洞大多數都能被實際利用來進行攻擊[2][3][16]。

### 3.1 Overflow / Underflow

- Overflow
  在 Solidity 中可以處理 256 bit 數字，最多為 $2^{256}$-1，溢位是系統給予的存放空間為有限之下，所需表示的數值超過範圍，因此在金流應用的合約中，無適當處理溢位問題將會造成非常大的損害，如下圖一所示。

```
  FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
+                                1
= 100000000000000000000000000000000
```

圖一：上溢位示意圖

如上圖所示，由於超出系統範圍，因此最大值加上 1 後系統將會判定錯誤，最終結果會返回 0。

- Underflow

  下溢位與上溢位同理，由於合約是圍繞著金流的應用做開發，因此在 Solidity 中並無負數的存在，因此最小值 0 減去 1 後將會造成下溢位的情形發生，如下圖二所示。

```
  0000000000000000000000000000000000
-                                1
= FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
```

圖二：下溢位示意圖

### 3.2 Out of gas

Gas 在 Solidity 扮演著非常重要的角色，他除了能讓程式設計師付出對應的手續費，區塊中的 gas limit 也能有效防禦惡意攻擊者使用高複雜度的程式碼使網路過度忙碌，導致他人交易速度延緩，然而當使用 Ethereum Wallet 上的 Send 函式功能，或是在合約中撰寫 send()、transfer()將 Ether 傳給其他合約時，卻可能發生 out of gas 異常，這是因為 address.send(amount)是一個空簽名的呼叫方式，此方式會觸發合約的 fallback function，而此時的 fallback function 被限制最大值為 2,300 個 gas，然而，2300 個 gas 能做的事十分有限，所有會改變合約狀態或需寫入區塊鏈的動作都將導致超過標準，如以下常見的 4 個動作：

- 改變合約內變數值
- 創建合約
- 呼叫內部、外部函式
- 發送 Ether

由於以上規則，好的程式設計師在撰寫合約時，fallback function 都會避免需消耗超過 2300 個 gas，以保證自己的合約可讓接受其他帳戶或合約傳送的 Ether，通常會撰寫 event 來簡單從區塊鏈讀取數據做為日誌使用。

另外，以太坊的每個區塊都有 gas limit 限制，以及每個合約也有該合約的 gas limit，當合約中使用了迴圈做撰寫則需特別注意，迴圈的堆疊次數很有可能會導致 gas 消耗量超越 gas limit，這會導致合約在某個點被強迫停止。

### 3.3 Call to the unknown

在 Solidity 中，當使用者呼叫外部函式時發生錯誤，例如參數的型態宣告錯誤或者參數名稱錯誤，導致該呼叫匹配不成功，進而觸發該外部函式的 fallback function，又或者使用者將 Ether 發送給某合約時，由於發送 Ether 是空簽名的函式，也將觸發合約的 fallback function，這個規則可能會合約意外停止或導向未知的地方，我們以合約「King of the Ether」來展示利用此漏洞進行攻擊的效果，以下將分 4 個步驟解說。

- 步驟 1

  首先撰寫並部屬簡易版國王合約，程式碼如下，我們將初始國王設定為合約擁有者，初始國王金額為 0.1 Ether，並將篡位金額設定為 200%，第 24 行的函式內容為將合約內金額轉送給當前國王，利用 onlyKing 修飾符來呼叫第 5 行的 modifier 功能，限定只有符合國王的 Address 才可執行該函式

```
1  contract KingOfEther {
2      address public king;
3      uint public king_Bid;
4
5      modifier onlyKing{
6          require(msg.sender == king);
7          _;
8      }
9
10     function KingOfEther() public payable{
11         king = msg.sender;
12         king_Bid = 0.1 ether;
13     }
14
15     function() public payable {
16         require(msg.value > king_Bid);
17         if(!king.call.value(king_Bid)()){
18             revert();
19         }
```

```
20              king = msg.sender;
21              king_Bid = msg.value * 2;
22          }
23
24          function getbalance() public onlyKing{
25              require(king.send(this.balance));
26          }
27      }
```

- 步驟 2
  接著我們假設了一個一般的篡位者叫作「NCHU」，該篡位者將至少投入初始國王金額的 200%，也就是 0.2 Ether 即可篡位成功，此時 NCHU 成為了新的國王，並且將國王金額補償給前國王後，新的國王金額也再度提升為 200%，也就是 0.4 Ether。
- 步驟 3
  接著，我們部屬一個攻擊合約「CallToTheUnknown」，程式碼如下，第三行的 BecomeKing 函示內容為傳送一筆金額給指定 address，而我們在第 10 行的 fallback function 加入了 revert()，此函式會回復所有交易動作。部屬完成後，執行 BecomeKing 函式，目標 address 為國王合約之 address，並輸入至少 0.4 Ether，新的國王將是合約帳戶 CallToTheUnknown，國王金額也將再度改變。

```
1   contract CallToTheUnknown {
2
3       function BecomeKing(address _address, uint Amount)public {
4               if(!_address.call.value(Amount)())
5               revert();
6       }
7
8       function CallToTheUnknown() public payable{}
9
10      function() public payable {
11              revert();
12      }
13  }
```

- 步驟 4
  接著，假設 NCHU 想再度篡位成為國王，以規則來說他只需要傳送大於國王金額的數目，我們嘗試傳送 1 Ether 給國王合約，此時將出現錯誤，這是因為此國王合約已經遭到合約 CallToTheUnknown 永久壟斷，當任何使用者輸入國王金額企圖篡位，就會觸發合約 CallToTheUnknown 的 fallback function 中的 revert 函式，任何的動作都會被還原，也就是說沒有人能再度篡位成功，合約 CallToTheUnknown 將永久成為國王。

### 3.4 Reentrance

此漏洞被稱之為重複攻擊或重複呼叫衝擊，在一般程式語言中，程式設計師能確信，當某函式為非遞迴函式，在無特殊例外情況下，該函式在終止前不能重新呼叫或執行，但是在 Solidity 中，由於 fallback function 的特殊規則，這可能導致一般函式在發送 Ether 時觸發了 fallback function 而導致類似遞迴的重複呼叫函式情況發生，這是在一般程式設計師的意料之外，這將可能造成大量的損失、如使該合約重複發送 Ether，本論文用合約「The DAO」來展示利用此漏洞進行攻擊的效果，以下將分 5 個步驟解說。

- 步驟 1
  首先撰寫並部屬簡易版銀行存款合約，程式碼如下，合約中第 15 行的 Deposit 為存款函式，使用者可輸入任意金額將之存入到合約內，並可使用第三的 mapping 功能以自己的帳戶 address 去查詢存款餘額。合約中第 5 行的 withdraw 函式為提款功能，使用者須輸入提款金額以及自己的帳戶或合約 address 後，通過金額判斷式即可提款成功，餘額將在提款完成後自動扣款。

```
1   contract Bank {
2
3       mapping (address => uint)public shares;
4
5       function withdraw(uint money, address user)public {
6       uint share = shares[user];
7       if(money < share){
8           if (!msg.sender.call.value(money)()){
9           revert();
10          }
11          shares[user] -= money;
12          }
13      }
```

```
14
15        function deposit()public payable {
16            shares[msg.sender] += msg.value;
17        }
18    }
```

- 步驟 2

  此步驟為測試功能，我們假設某帳戶 Main Account 執行 Deposit 函式，並存入 100 Ether，此時合約總金額以及使用 address 查詢該帳戶存款金額皆為 100 Ether。

- 步驟 3

  接著，我們部屬一個攻擊合約 DAOattack 如下，合約一開始為 Bank 合約的原始碼，目的是為了從外部合約呼叫本合約的函式，並且在第 3 行以及第 8 行的初始化函式作宣告，第 8 行的輸入參數 DAO 即是合約 Bank 的 address，如此即可使用 B.function() 的方式來呼叫合約 Back 之函式。第 12 行 attack 函式為攻擊函式，他會呼叫 Back 合約的 withdraw 函式，輸入參數為變數 attackfond 以及攻擊者的 address，attackfond 在第 5 行宣告為 1 Ether，是使用重複進入攻擊時每回合的盜取金額，執行後提款成功將會因為 Bank 合約發送金額給攻擊合約而觸發攻擊合約的 fallback function，合約在 fallback function 中寫入一個迴圈，將導致能從 Bank 合約重複提款直到迴圈停止。

```
… //Bank contract
2   contract DAOattack{
3       Bank B;
4       uint num;
5       uint attackfond = 1 ether;
6       address owner = msg.sender;
7
8       function DAOattack(address DAO) public payable {
9           B = Bank(DAO);
10      }
11
12      function attack()public {
13          B.withdraw(attackfond, owner);
14      }
15
16      function ()public payable {
```

```
17              num++;
18              if(num < 30) B.withdraw(attackfond, owner);
19          }
20
21          function getBalance() public constant returns(uint) {
22            return this.balance;
23           }
24      }
```

- 步驟 4
  由於我們在部屬攻擊合約時，宣告的 attackfond 為 1 Ether，這代表當攻擊者執行提款函式時，攻擊者的餘額不能少於 1 Ether，於是攻擊者偽裝為一般存款人向 Bank 存款，假設為 2 Ether，接著攻擊者去 Bank 合約中查詢餘額狀況，攻擊者的餘額將為 2 Ether，Bank 合約的總金額也將提升。
- 步驟 5
  確認餘額狀況後，即可執行攻擊合約的 attack 函式，該函式將因為 Reentrance 漏洞而重複執行提款函式直到迴圈結束，此時查看攻擊者的合約，可以看到總金額將變成了 30 Ether，正好是迴圈數(30)乘上變數 attackfond(1 Ether)，代表攻擊者成功的只用了 2 Ether 的存款，卻提款出了 30 Ether。

### 3.5 Visibility / Privacy

在撰寫合約時，函式的撰寫可以選擇 public、private 等修飾符來設定是否公開，使用者可以設定為 public 讓該函式可於 Ethereum Wallet 供他人讀取或執行，而當設定為 private 其他使用者則無法從合約介面讀取或任何方式呼叫，但這仍無法保護其隱私性，這是因為任何需要寫入區塊鏈的動作，如改變合約中的變數值，皆需要提出交易並等待礦工們驗證，礦工變會將若干交易發佈至區塊鏈網路中，然而區塊鏈網路是完全公開的，任何人都能讀取區塊鏈中的任何數據或資料，其中當然也包含了擁有 private 修飾符的函式內容。本論文用合約「Guess Number」來展示利用此漏洞進行攻擊的效果，以下將分 3 個步驟解說。

- 步驟 1
  撰寫及部屬合約，程式碼如下，第 13 行 play 函式可以讓使用者輸入一個數字，並投入 1 Ether，使用者如投入其他金額將會返回動作，輸入後會把使用者的 address 及輸入的數字存入 Players 中，函式後段當變數 num 等於 2(代表已有兩位使用者完成 play 函式)，將執行 Winner 函式，Winner 函式將會判斷兩位使用者輸入數字的總和，並判斷為偶數或基數，如為偶數，使用者 1 獲勝並收到獎勵 1.8 Ether，反之則

為使用者 2 獲勝，差額的 0.2 Ether 儲存在合約中作為合約擁有者的手續費。第 31 行 getProfit 函式為合約擁有者才能執行成功，可以把合約中的所有手續費轉到自己帳戶 address 中。

```
1  contract GuessNumber{
2      struct Player {
3          address addr;
4          uint number; }
5      Player[2] private players;
6      uint num;
7      address owner;
8
9      function GuessNumber () public{
10         owner = msg.sender;
11         }
12
13     function play(uint number) public payable{
14         if (msg.value != 1 ether)    revert();
15         players[num] = Player(msg.sender, number);
16         num++;
17         if (num==2) Winner();
18         }
19
20     function Winner() private {
21         uint n = players[0].number+players[1].number;
22         if (n%2==0)
23             players[0].addr.transfer(1.8 ether);
24         else
25             players[1].addr.transfer(1.8 ether);
26         delete players;
27         num=0;
28         }
29
30     function getProfit() public{
31         if(msg.sender!=owner)    revert();
32         msg.sender.transfer(this.balance);
```

```
33        }
34    }
```

- 步驟 2
  假設第一位使用者 A 執行了 play 函式並輸入 1 Ether 與數字 10，驗證完成後，使用者 2 可從區塊鏈網路中尋找此筆資料，可以很明確的看到此筆交易的 gas used、gas limit、發起交易者的 address 等等，當然也能看到使用者 A 的輸入參數。
- 步驟 3
  此時，使用者 B 只要利用此特性，在執行 Play 函式前先至區塊鏈網路中查詢，他可以永遠的知道前一位使用者的輸入參數，以此為例，使用者 B 只需輸入任意基數就可以獲取贏家獎勵 1.8 Ether。

## 肆、結論

在本節中，本論文將分析各個漏洞或弱點的解決或避免方式。

### 4.1 Overflow / Underflow

在撰寫合約如果會有益位的情況發生，可以在合約內導入 OpenZeppelin 的 SafeMath library，該程式碼如下。

```
1   library SafeMath {
2
3       function mul(uint256 a, uint256 b) internal pure returns (uint256 c) {
4           if (a == 0)
5               return 0;
6           c = a * b;
7           assert(c / a == b);
8           return c;
9       }
10
11      function div(uint256 a, uint256 b) internal pure returns (uint256) {
12          return a / b;
13      }
14
```

```
15      function sub(uint256 a, uint256 b) internal pure returns (uint256) {
16          assert(b <= a);
17          return a - b;
18      }
19
20      function add(uint256 a, uint256 b) internal pure returns (uint256 c) {
21          c = a + b;
22          assert(c >= a);
23          return c;
24      }
25  }
```

以上 4 個函式分別為乘法；除法；減法及加法，在撰寫合約時，可以在自己的合約上方加入此 Library，當導入 SafeMath library，4 種運算函示會自動判斷是否有溢位的情況，如有溢位發生會拋出錯誤。

### 4.2 Out of gas

在以太坊中發送 Ether 時，假設沒有 Ether 不足而發送失敗，有可能發生的情況歸類如下。

- 合約→合約
  - 發送成功，因為接收方合約的 fallback function 消耗低於 2300 個 gas。
  - 發送失敗，因為接收方合約的 fallback function 消耗高於 2300 個 gas。
  - 發送成功，因為傳送方使用 address.call.value(Amount)()的方式發送 Ether，此方式不受 2300 個 gas 的限制。
  - 發送失敗，因為合約內沒有寫入 fallback function 並加上修飾符 payable。
- 合約→帳戶
  - 發送成功
- 帳戶→帳戶
  - 發送成功

從以上歸類情況可以看出，發送 Ether 時的接收方為合約時才有發送失敗的情況出現，因此合約內的 fallback function 規則及撰寫方式須多加留意。

### 4.3 Call to the unknown

以合約 King of the Ether 為例，因為將補償傳送給國王時，當國王為一個合約，將

觸發的 fallback function 而導致不明狀況發生，因此在合約傳送金額時應採取一些防護措施而預防此情況發生，以下將合約 King of the Ether 的 fallback function 更改如下。

```
1        function() public payable {
2            bool res;
3            require(msg.value > king_Bid);
4            res = king.call.value(king_Bid)();
5            king = msg.sender;
6            king_Bid = msg.value * 2;
7        }
```

如上述程式碼，我們以第 4 行的方式來取代原本的第 17-18 行，這樣的方式即使第四行的發送結果為失敗，也不影響後續程式的執行，而發送失敗的 Ether 將保留在合約內供下個國王轉移，因此嘗試使用漏洞的攻擊者除了被篡位成功，還無法拿到任何的補償。

### 4.4 Reentrance

在 The DAO 合約中，因為 withdraw 函式的漏洞而導致使該函式能重複執行，因此本論文提供了三種方式來預防此漏洞。

- 使用 transfer()或者 send()來預防 Reentrance 攻擊，因為這兩個函式皆受到 2300 個 gas 的限制。
- 使用 address.call.value(etherAmount).gas(gasAmount)，gasAmount 可自訂此程式執行能消耗的最大 gas 數量。
- 將更新餘額程式碼的順序擺放至發送提款之前，如以下所示。

```
1        function withdraw(uint money, address user)public {
2        uint share = shares[user];
3        if(money < share){
4        shares[user] -= money;
5            if (!msg.sender.call.value(money)()){
6                revert();
7                }
8            }
```

```
9       }
```

如果以上述兩種使用限制 gas 數量之方式，執行提款的合約將須注意 fallback function 的撰寫方式，過於複雜或惡意的程式碼都將造成提款失敗。

### 4.5 Visibility / Privacy

由於以太坊平台本上就是建立在區塊鏈技術底下，所以並沒有任何方法可以改變區塊鏈的公開性、透明性問題，這將影響許多賭博性質的合約、如樂透、彩票等等。在 Solidity 中，產生隨機數字是相對困難的，不過以 GuessNumber 合約為例，合約中判斷的方式僅為簡單的偶數或基數，因此可以使用以下函式來使挑戰者贏取獎勵的機率更為公平。

- block.difficulty
- block.gaslimit
- block.timestamp

以上三種內建函式皆會返回一個使用者無法自行控制的 uint 型態數字，以 block.timestamp 為例，使用者並無法確定自己發出的交易會在何時被礦工確認，以 GuessNumber 合約來說，以擁有足夠的隨機性。

## 參考文獻

[1] M. Y. Afanasev, Y. V. Fedosov, A. A. Krylova and S. A. Shorokhov, "An application of blockchain and smart contracts for machine-to-machine communications in cyber-physical production systems," *2018 IEEE Industrial Cyber-Physical Systems (ICPS)*, pp. 13-19, IEEE, May 2018.

[2] N. Atzei, M. Bartoletti and T. Cimoli, "A survey of attacks on ethereum smart contracts (sok)," *International Conference on Principles of Security and Trust*, pp. 164-186, Springer, Berlin, Heidelberg, 2017.

[3] S. Bragagnolo, H. Rocha, M. Denker and S. Ducasse, "SmartInspect: solidity smart contract inspector," *2018 International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, pp. 9-18, IEEE, Mar. 2018.

[4] V. Buterin, "A next-generation smart contract and decentralized application

platform," *Ethereum white paper*, 2014.

[5] Y. H. Chen, S. H. Chen and I. C. Lin, "Blockchain based smart contract for bidding system," *2018 IEEE International Conference on Applied System Invention (ICASI)*, pp. 208-211, IEEE, Apr. 2018.

[6] C. Dannen, *Introducing Ethereum and Solidity*, Berkeley: Apress, 2017.

[7] E. Hildenbrandt, M. Saxena, X. Zhu, N. Rodrigues, P. Daian, D. Guth and G. Rosu, "Kevm: A complete semantics of the ethereum virtual machine," Aug. 2017.

[8] Y. Hirai, "Defining the ethereum virtual machine for interactive theorem provers," *International Conference on Financial Cryptography and Data Security*, pp. 520-535, Springer, Cham, Apr. 2017.

[9] A. Judmayer, N. Stifter, K. Krombholz and E. Weippl, "Blocks and Chains: Introduction to Bitcoin, Cryptocurrencies, and Their Consensus Mechanisms," *Synthesis Lectures on Information Security, Privacy, & Trust*, *9*(1), pp. 1-123, 2017.

[10] D. Y. Kao and S. C. Hsiao, "The dynamic analysis of WannaCry ransomware," *2018 20th International Conference on Advanced Communication Technology (ICACT)*, pp. 159-166, IEEE, Feb. 2018.

[11] S. Mohurle and M. Patil, "A brief study of wannacry threat: Ransomware attack 2017," *International Journal of Advanced Research in Computer Science*, vol.8, issue 5, May 2017.

[12] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," 2008.

[13] A. Sudhan and M. J. Nene, "Employability of blockchain technology in defence applications," *2017 International Conference on Intelligent Sustainable Systems (ICISS)*, pp. 630-637, IEEE, Dec. 2017.

[14] D. Tse, B. Zhang, Y. Yang, C. Cheng and H. Mu, "Blockchain application in food supply information security," *2017 IEEE International Conference on Industrial Engineering and Engineering Management (IEEM)*, pp. 1357-1361, IEEE, Dec. 2017.

[15] G. Wood, "Ethereum: A secure decentralised generalised transaction ledger," *Ethereum project yellow paper*, *151*, pp. 1-32, 2014.

[16] E. Zhou, S. Hua, B. Pi, J. Sun, Y. Nomura, K. Yamashita and H. Kurihara, "Security Assurance for Smart Contract," *2018 9th IFIP International Conference on New*

*Technologies, Mobility and Security (NTMS)*, pp. 1-5, IEEE, Feb. 2018.