# Analiza scenariuszy zaburzeń systemu

## Dominacja obliczeniowa jednego węzła

Występuje w sytuacji, gdy jeden węzeł jest w stanie wykopać blok z dużą większą prędkością niż pozostali kopacze.

Scenariusz jest realizowany poprzez dodanie dodatkowej trudności (parametr DIFFICULTY + 2) do wymagań kopania dla każdego węzła oprócz 5001, który wciela się w rolę dominatora. Węzły wykopane przez niego są wciąż poprawne, ponieważ trudność wymagana przy weryfikacji bloku nie ulega zmianie.

### Przypadek bez dominatora

Do porównania wyników wykorzystano wizualizacje oraz liczność bloków wykopanych przez różne węzły co można sprawdzić odpytując wybrany węzeł o listę bloków, grupując je według kopacza i przedstawiając liczebności (program gb.js do tego grupowania został wygenerowany przez Copilot i sprawdzony osobiście – było to możliwe ponieważ jest relatywnie prosty). Przy zwykłej pracy sieci:

Miner Counts: { '5001': 5, '5002': 5, '5003': 3, '5004': 5, undefined: 1 }

Undefined to blok GENESIS. W pozostałych przypadkach widać, że różni kopacze wykopali podobne ilości bloków. W skrypcie wysyłane jest 9 transakcji, a suma jest dużo większa – jest to skutek losowo tworzonych forków, które nie przeszkadzają w badaniu.

### Przypadek uczciwy

* Trudność dla dominatora: 2
* Trudność dla kopaczy: 2 + 2

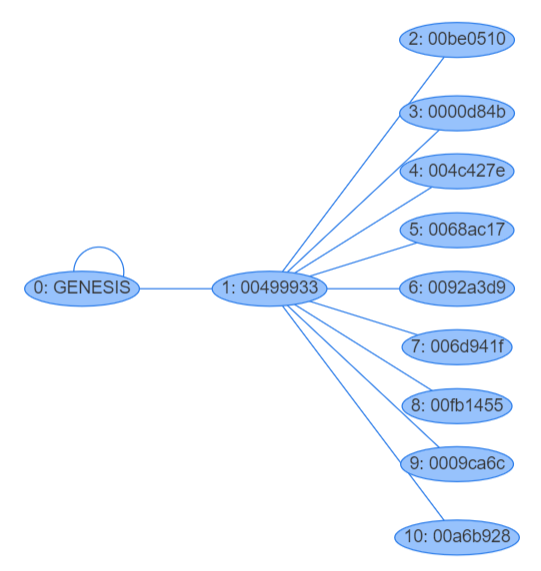
Przypadek w którym jeden węzeł dominuje, ale jest uczciwy sprawia, że wszystkie bloki są wykopane przez niego. Wynikiem przykładowego skryptu jest liczność bloków:

Miner Counts: { '5001': 9, undefined: 1 }

Łatwo zauważyć dominację 5001 oraz fakt, że nie dochodzi do żadnych forków. W przypadku założenia, że kopacz nie dostaje nagrody za dany blok taka sytuacja nie ma większego znaczenia, natomiast w prawdziwej sieci sprawiło by to, że 5001 bogacił by się znacznie szybciej niż inni. Mogło by to spowodować zarówno, że pozostali zrezygnowali by z Coina powodując jego upadek, ale mogło by to też zadziałać jako motywacja do zwiększenia możliwości obliczeniowych w celu rywalizacji.

### Złośliwy węzeł dominujący

Test zakładał włączenie złośliwości 5001 w postaci celowego rozwidlania łańcucha (kopie nowy blok nie dla poprzedniego bloku będącego końcem łańcucha tylko dla przedostatniego). Liczebności wykopanych bloków są takie same jak w poprzedniej podsekcji. To co jest ważne to wygląd łańcucha:



Powstały w tym scenariuszu łańcuch pokazuje pełną kontrolę węzła nad siecią. Żaden inny węzeł nie ma szansy wykopać bloku, który wydłuży łańcuch bo brakuje mu mocy, a węzeł dominujący celowo tego nie robi – oznacza to **całkowite zatrzymanie sieci**!

Ryzyko całkowitego zniszczenia sieci zależy od rozmiaru i parametrów trudności dla waluty. W bardzo rozległym systemie z odpowiednio krótkim czasem kopania istnieje szansa, że węzły-sąsiedzi tego, który inicjalizuje transakcje byli by w stanie wykopać ją szybciej niż rozpropagowała by się do daleko położonego dominatora. Nie zmienia to faktu, że nawet w optymistycznym przypadku sieć i jej działanie było by **zaburzone**.

### Propozycja rozwiązania

Najlepszą metodą wydaje się być znaczne **zwiększenie trudności hasha**. Dla bardzo wysokiej trudności element losowości przy poszukiwaniu będzie miał dużo większe znaczenie niż dla łatwych hashy, gdzie przeszukanie wszystkich kombinacji jest błyskawiczne. Spośród innych wymyślonych metod żadna nie wydawała się nawet podobnie skuteczna, co ta.

W celu przetestowania zmienione zostały parametry trudności:

* Trudność dla dominatora: 5
* Trudność dla kopaczy: 5 + 2

Miner Counts: { '5001': 7, undefined: 1 } (zatrzymane wcześniej z powodu czasochłonności)

Niestety nie pomogło to w badanym problemie. Hipotezą jest to, że **trudność jest wciąż zbyt mała**, a **dysproporcja między węzłami** (trudność +2) wręcz **nierealnie duża**, ale po zmniejszeniu jej do +1 dominacja nie była zbyt zauważalna nawet dla mniejszych trudności. Żeby upewnić się, że metoda rozwiązania ma sens należałoby sprawdzić jeszcze większe trudności, bądź zastosować inną metodę spowalniania innych węzłów np. poprzez krótki timeout na każdą próbę jednego nonce.

## Awaria węzła przegubowego

Scenariusz to sytuacja, w której awarii ulega połączenie typu most. Prowadzi to do utworzenia dwóch podsieci, które mogą kopać zupełnie inne transakcje. Test odbył się poprzez utworzenie sieci, gdzie 5002 jest węzłem przegubowym mającym dwa mosty. Pauzowanie go, czyli wyłączanie odpowiedzi na jakikolwiek broadcast symulowało awarię. W testach wykorzystywane są endpointy leave\_network i pause – więcej o nich w sekcji o nieautoryzowanej kontroli.

### Kontrolowane odłączenie

Test polegał na przetestowaniu działanie mechanizmu kontrolowanego odłączenia. Kiedy węzeł otrzyma sygnał wykonuje następujące czynności:

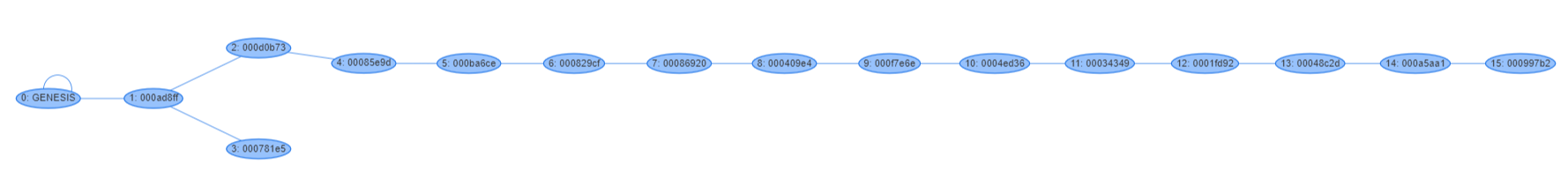
* Odpytuje swojego mastera (adres do którego dołączył na początku), czy może opuścić sieć – ma to na celu zapobiec sytuacji, w której sąsiedzi są przekierowywani do nowego węzła, który zaraz opuści sieć
* Po otrzymaniu zezwolenia wysyła wszystkim sąsiadom adres nowego mastera
* Węzeł się zatrzymuje

Metoda jest skuteczna i dobrze zapobiega powstawaniu nadmiarowych połączeń, ale niestety jest podatna na oszustwo. **Złośliwy węzeł master mógłby nie pozwalać na odłączenie się węzła**. Żeby temu zapobiec odłączający węzeł mógłby rozesłać wszystkim pełną listę sąsiadów, ale spowodowało by to tylko nadmiarowe połączenia a w przypadku, gdy jest to węzeł przegubowy, a jego master też się odłącza to **sieć została by rozcięta**. W aktualnej implementacji niestety żadna wersja nie jest w 100% pewna. Potencjalne rozwiązania w dalszych sekcjach.

Test w uczciwym scenariuszu polegał na:

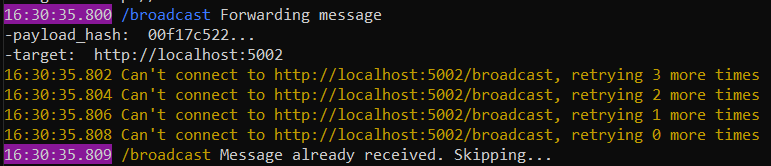
* Ustawienie sieci
* Wysłanie 2 transakcji startowych
* Wyłączenie węzła 5002 (bez systemu odłączania powodowałoby dziurę w sieci)
* 7 transakcji od 5001
* 5 transakcji od 5003

Po zakończeniu skryptu blockchain był identyczny dla obu 5001 i 5003, co potwierdza, że **mechanizm zadziałał w tym scenariuszu**.

  
Blockchain po badaniu (identyczny dla obu sieci)

### Trwałe wstrzymanie przekazywania danych

Podstawowa symulacja awarii pokazała reakcje okolicznych węzłów. Poniższy zrzut ekranu z węzła 5001 pokazuje, że **utrata połączenia została wykryta**. W aktualnej implementacji węzeł nie reaguje na to ostrzeżenie, więc **sieć przestaje działać**. Dodanie kodu, który spowodowałby kontrolowane wyłączenie się węzła z sieci i zwrócenie ostrzeżenia użytkownikowi po danej liczbie prób połączenia byłby jednak prosty do dodania.

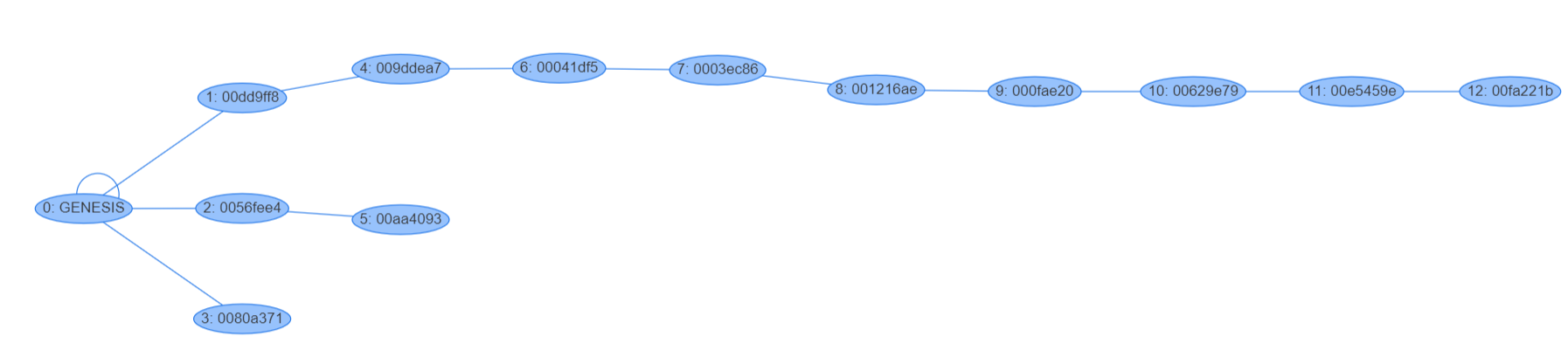
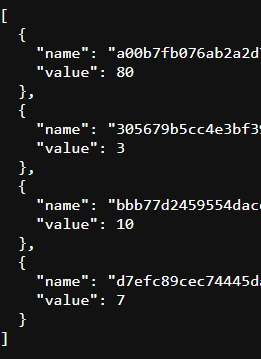
  
Zrzut ekranu z logów węzła 5001

### Tymczasowe wstrzymanie przekazywania danych

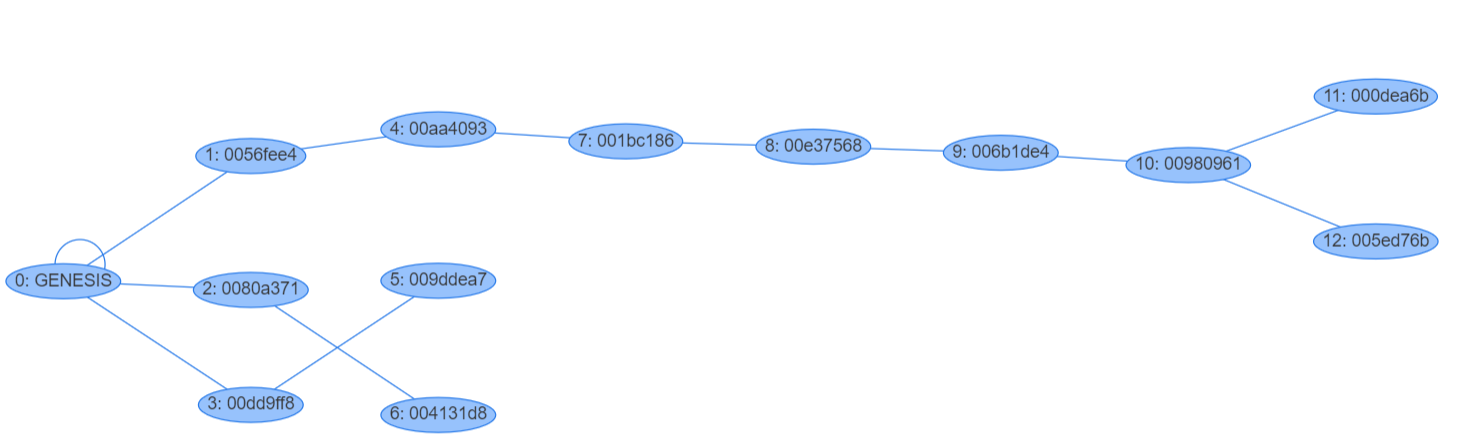
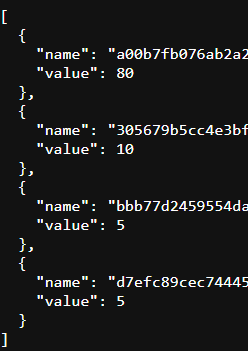
Ten scenariusz działa analogicznie do poprzedniego, z tą różnicą że węzeł przegubowy wraca po pewnym czasie. Zrealizowano to wykorzystując endpoint /pause węzła 5002. Scenariusz testu to:

* Utworzenie sieci
* Wysłanie 2 transakcji zasilających konta 5001 oraz 5003
* Wyłączenie mostu
* Wysłanie 7 transakcji dla podsieci 1
* Wysłanie 5 transakcji dla podsieci 2
* Włączenie mostu
* Wysłanie kilku (3) transakcji wywołujących mechanizm synchronizacji

Stan przed włączeniem mostu (do przedostatniego kroku włącznie).

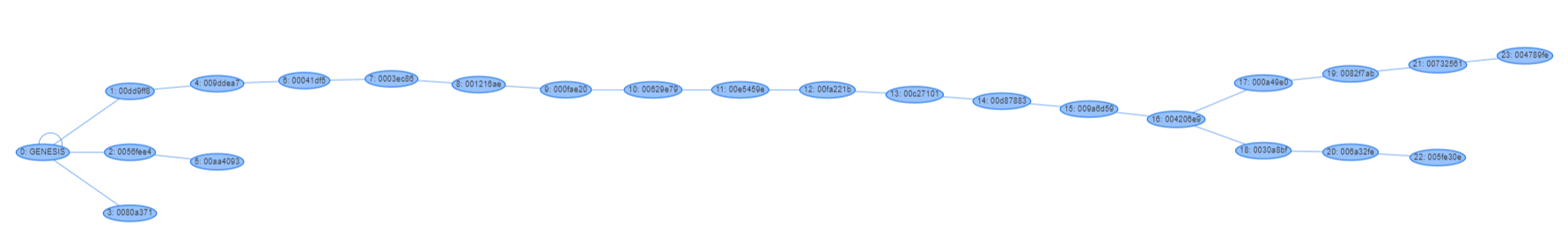
Podsieć 1 i stany kont (węzeł 5001)

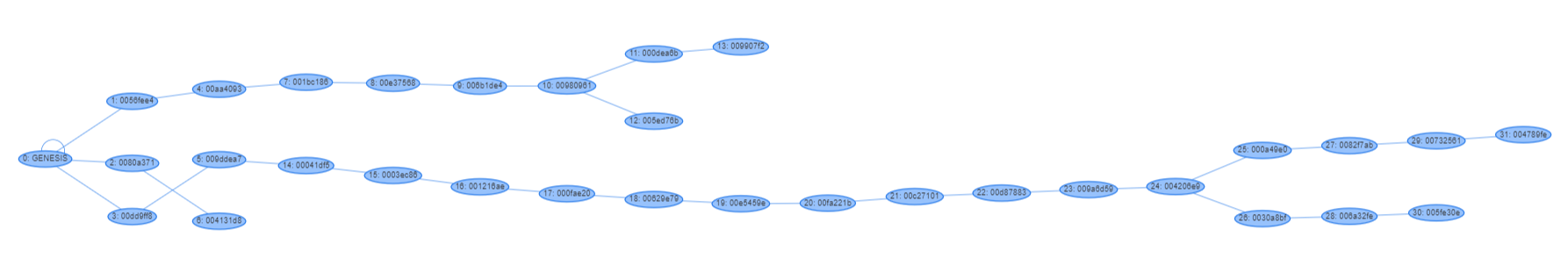
  


Podsieć 2 i stany kont (węzeł 5003)

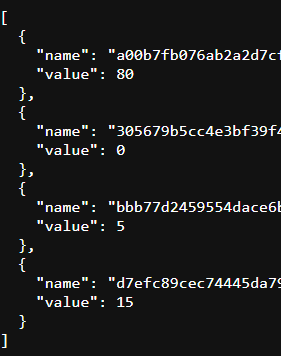
Można zaobserwować, że zaakceptowane (najdłuższe) łańcuchy są inne. Różnią się też długością – w podsieci 1 długość to 2 (wstępne zasilanie) + 7 (transakcje w podsieci 1), natomiast dla podsieci 2 jest to 2 + 5.

Po włączeniu mostu:

  
Podsieć 1 (5001) – blok końcowy 004789fe

  
Podsieć 2 (5003) – blok końcowy 004789fe

Dłuższy łańcuch podsieci wygrał i został zaakceptowany jako ten ważniejszy. Aby nie naruszać spójności sieci, wszystkie transakcje z podsieci 2 zostały wycofane i wykopane ponownie, dlatego długość nowego wspólnego łańcucha to 17 (2 wstępne + 7 podsieć 1 + 5 podsieć 2 + 3 sychronizujące). Stany kont sprawdzone po synchronizacji również były zgodne dla obu węzłów i poprawne biorąc pod uwagę skrypt rozsyłający:

  
Stany kont po synchronizacji (te same dla wszystkich węzłów)

Można więc powiedzieć, że sieć jest **dobrze przygotowana** na ten konkretny scenariusz.

### Złośliwy węzeł

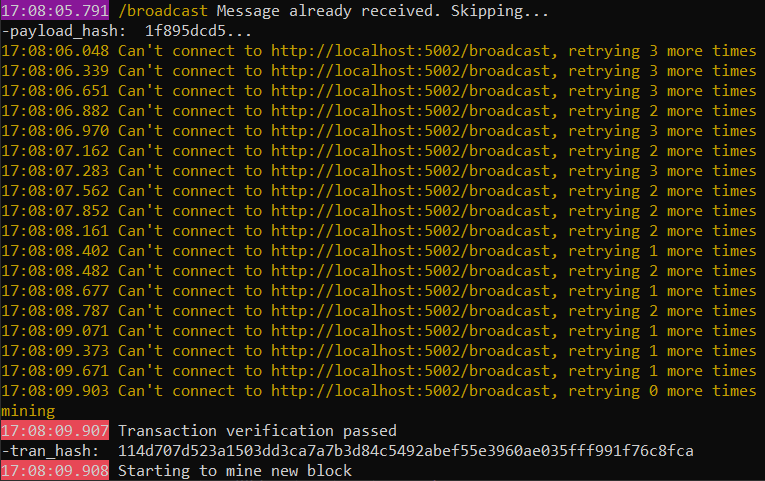
Złośliwość mogłaby polegać na wyłączeniu przekazywania wiadomości broadcast, ale w taki sposób by nie wywołać wykrycia awarii. W ramach testu odpowiedź pauzowanego węzła została zmodyfikowana tak, aby nie powodować błędów. W tym scenariuszu wszystko zadziałało tak samo, ale podsieci nigdy **nie dowiedziały się o awarii 5002**, co oznacza że jego złośliwość mogłaby spowodować **rozerwanie sieci na pomniejsze**.

### Problem nieskończonej odpowiedzi

W przypadku, gdy węzeł nie odpowiada to istnieje ryzyko zablokowania łącza i nieskończone oczekiwanie. Problem ten został napotkany w fazie tworzenia projektu i zabezpieczony parametrem metody fetch:

signal: AbortSignal.timeout(1500)

Parametr mówi, że połączenie zostanie porzucone po 1.5s bez odpowiedzi. Ustawienie to jest jednak dość nadmiarowe i prawdopodobnie mogłoby być mniejsze.

  
Zrzut logów z 5001

Wadą zaimplementowanego loggera jest to, że pokazuje czas wyświetlenia wiadomości, niekoniecznie wydarzenia, a przy wielu asynchronicznych zapytaniach czasem powstaje tłok. To co jest ważne na zrzucie to to, że mimo nieudanych połączeń praca była kontynuowana, czyli **ten problem został zabezpieczony.**

### Propozycja rozwiązania

Przy awarii łącza sytuacja jest dość prosta do rozwiązania. Awarię można rozpoznać poprzez wykrycie braku odpowiedzi na wysyłane do węzła wiadomości broadcast. Można również dla pewności okresowo wysyłać do sąsiadów prosty ping w celu diagnostycznym. W sytuacji utraty siedzi następnie należało by poczekać chwilę i spróbować ponownie, bądź od razu zwrócić użytkownikowi błąd i poprosić o ponowne dołączenie do sieci – tym razem podając inny adres docelowy połączenia.

W przypadku złośliwego węzła sytuacja się komplikuje:

Pierwszą sugestią jest tworzenie określonej liczby nadmiarowych połączeń przy dołączaniu do sieci. Dla poprawnej sieci jedno połączenie jest wystarczające jednak w celu zabezpieczenia można dołączyć np. do 3 węzłów, które już są w sieci. Sprawi to, że powstaną połączenia zapasowe, ale kosztem zwiększonej redundancji ruchu w sieci. Liczbę nadmiarowych połączeń można by było dobrać eksperymentalnie w celu uzyskania balansu. Ważne jest też, aby nadmiarowe połączenia prowadziły do jak najbardziej odległych węzłów, aby zminimalizować ryzyko że będą one w zmowie.

Drugą metodą jest okresowe wysyłanie sygnału przez węzeł główny INIT. Mógłby on rozsyłać raz na np. 5 minut wiadomość zaszyfrowaną kluczem prywatnym asymetrycznym, którą po odbiorze inne węzły rozszyfrowały by w celu weryfikacji, czy na pewno pochodzi od INIT. Jeśli wiadomość by nie docierała przez wyspecyfikowany czas to węzeł wiedziałby, że stracił połączenie. Nie jest to jednak pomysł warty dalszych rozważań, ponieważ daje „pole do popisu” dla złośliwych węzłów, które blokowały by broadcast, ale przepuszczały sygnał kontrolny.

## Nieautoryzowana kontrola nad węzłami

### Endpointy testowe - /pause, /leave\_network i inne

Pauzowanie to mechanizm powodujący zatrzymanie odpowiedzi węzła na broadcast. Opuszczanie sieci to kontrolowany sposób wyłączania węzła. Oba te endpointy są możliwe do użycia przez każdego atakującego – jest to jednak świadome i oznaczone. Służą one ułatwieniu dostępu i skryptowania w ramach testu. W ostatecznej wersji realnego programu (nie tylko proof of concept) należało by je usunąć i ewentualnie zastąpić czymś dostępnym przez GUI bądź konsolę z poziomu użytkownika.

Program zawiera też kilka endpointów testowych do odczytu danych, które np. zwracają listę sąsiadów. Jest to oczywiście naruszenie poufności – sytuacja w tym przypadku jest analogiczna co do poprzedniej.

Podsumowując, istnieją endpointy umożliwiające **nieautoryzowany dostęp**, ale są one **wykorzystywane do testów** i **oznaczone jako niebezpieczne**.

### Kontrolowane odłączenie

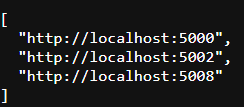
Test polegał na sprawdzeniu jakie skutki można wywołać ręcznie wysylając zapytania na endpointy odpowiedzialne za odłączenie.

#### DELETE /neighbors oraz PUT /neighbors

Poprzez wywołanie Powershell:

Invoke-WebRequest -Uri http://localhost:5001/neighbors -Method Delete -Body "[`"http://localhost:5008`"]" -ContentType "application/json"

Udało się doprowadzić do dodania niestniejącego węzła do listy sąsiadów:

  
Sąsiedzi 5001 po teście

Oznacza to, że endpoint umożliwia **tworzenie nadmiarowych i fałszywych połączeń** złośliwym aktorom. Metoda PUT pozwala na analogiczne działanie, ale trochę innym wywołaniem.

#### Inne

Ważnym faktem opartym o analizę endpointów związanych z kontrolowanym odłączeniem jest to, że **nie da się doprowadzić do usunięcia** żadnego węzła z listy sąsiadów. Jeśli węzeł faktycznie opuszcza sieć to będzie wywoływać awarie połączenia – inny mechanizm usuwający nieaktywne węzły spośród sąsiadów zajął by się tym problemem. **Pod kątem usuwania dostępu innym implementacja jest bezpieczna.**

## Nieuczciwy węzeł

### Akceptacja wszystkich transakcji

Test polegał na wysłaniu trzech niepoprawnych transakcji, a następnie jednej prawdziwej. W teście oszustem był 5001.



Bloki oszusta po teście (blok 4 odpowiada legalnej transakcji, 1-3 niepoprawnym)



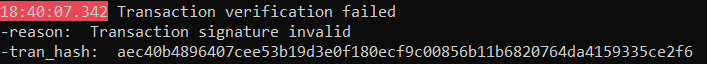
Bloki uczciwego kopacza po teście (blok 1 odpowiada legalnej transakcji)

Można zauważyć, że uczciwi kopacze dzięki weryfikacji bloków i transakcji nie dopuszczają nawet poprawnie wykopanych niespójnych transakcji. Ciekawym zjawiskiem jest to, że oszust nie przyjął poprawnego bloku wykopanego przez kogoś innego – jest tak, ponieważ nie węzły nie akceptują przestarzałych bloków. Można uznać, że **sieć jest odporna na tak prosty scenariusz złośliwości**.

### Podmiana wartości w transakcji

Podmiana odbywała się na etapie przetwarzania transakcji, po jej zaakceptowaniu złośliwy węzeł 5001 podwajał wartość przesyłanej waluty i podmieniał hash transakcji na nowy.

W teście prawdziwa transakcja była rozsyłana z węzła 5001.

  
Zrzut logów z węzła 5002

Jak widać weryfikacja transakcji nie powiodła się i transakcja nie została rozesłana. Błędem jest nieprawidłowość podpisu – oczywiście oszust nie może podpisać transakcji bez dostępu do czyjegoś klucza prywatnego, a jeśli ma do niego dostęp to może robić co chce bez „zabawy” w podwajanie i rehashowanie.

Ze względu na fakt, że sprawdzany jest hash całego zestawu danych i jego podpis to wiadomo, że **system jest odporny na jakiekolwiek podmiany danych transakcji**.

### Podmiana wartości bloku

Proces analogiczny do podmiany w transakcji, ale ze względu na umiejscowienie przetwarzania bloków lepiej było przeprowadzać testy dla węzła 5000 jako kopacza (zamiast 5001, 5004 jako kopacz bez zmian). Sprawdzone zostały scenariusze:

* zmiany danych
* zmiana danych i rehash
* zmiana danych i wstawienie łańcucha znaków z odpowiednią ilością zer jako hasha

Żaden z nich się nie powiódł. Mimo niedużej liczby testów wyczerpują one temat podmiany danych i pokazują **bezpieczeństwo systemu w tym kontekście**.

### Podmiana podpisu na własny

### Podszycie się pod coinbase

## Działanie w zmowie

## Ataki na portfel jako bazę danych

### Atak na łańcuch przechowujący klucz prywatny w bazie

Według dokumentacji użytej biblioteki (<https://nodejs.org/api/crypto.html#cryptocreateprivatekeykey> ), przy podaniu parametrów cipher oraz passphrase do klucza prywatnego przy generacji sprawia to, że przy konwersji na łańcuch jest on szyfrowany zgodnie ze standardem PKCS 5 (v2) (<https://datatracker.ietf.org/doc/html/rfc2898> ) co oznacza m.in użycie soli, wieloiteracyjnej funkcji wyprowadzenia klucza na bazie hasła oraz wyprowadzenia zarówno klucza ostatecznego i wektora IV dla AES z wyniku wspomnianej funkcji. Ze względu na skomplikowany charakter przetwarzania testy są bardzo trudne w realizacji. Można jednak przeprowadzić test polegający na sprawdzeniu, czy klucz prywatny w bazie zgodnie z założeniem jest przechowywany w sposób zaszyfrowany:

  
Przycięty zrzut z konsoli dla różnych metod wyświetlania klucza prywatnego

Z testu wynika, że łańcuch przechowywany w bazie jest zgodny z szyfrowaną wersją. Niestety dokładniejsze testy takie jak np. sprawdzenie kodu źródłowego i odtworzenie go poszczególnymi funkcjami w celu porównania wyniku nie były możliwe, ponieważ biblioteka korzysta z implementacji C++ dla bardziej wymagających obliczeniowo elementów takich jak szyfrowanie, a ze względu na brak wiedzy na temat tego języka analiza tak trudnego tematu nie jest możliwa.

Źródło pliku odpowiadającego za export kluczy: <https://github.com/nodejs/node/blob/main/lib/internal/crypto/keys.js>

Informacja o braku dostępu do funkcji C++, z których korzysta powyżej załączony keys.js:  
<https://github.com/nodejs/help/issues/3079?spm=a2c6h.13046898.publish-article.12.7f356ffaNSXXIp>

Przy tych testach jedyne co można powiedzieć to, że **klucz jest przechowywany poprawnie** więc **integracja z sqlite działa**, ale z samym bezpieczeństwem klucza **należy zaufać twórcom biblioteki** (poprzez „należy” rozumie się brak innego wyboru oprócz własnej implementacji – co nie jest ani łatwe, ani nie daje pewności skuteczności).

### Utrata danych logowania

Jeśli użytkownik straci dane logowania to jedyne do czego może uzyskać dostęp to identyfikator konta, klucz publiczny oraz zaszyfrowany klucz prywatny. Zaszyfrowana postać klucza skutkuje tym, że tak naprawdę jedyną możliwością odzyskania środków jest BruteForce, bądź znalezienie kolizji identyfikatora. Ze względu na użyte funkcje szyfrowania i skrótu jest to statystycznie praktycznie niemożliwe zakładając, że biblioteka kryptograficzna działa.

## Błędne żądania

Celem jest zbadanie zachowania węzłów w reakcji na niepoprawne dane żądań oraz ich nielogiczne wywołania. Reakcja nie powinna powodować uzyskania przez atakującego żadnych dodatkowych informacji, a praca węzłów nie powinna być zakłócona

|  |  |  |
| --- | --- | --- |
| Endpoint | Zmiana | Obserwacje dotyczące odpowiedzi i zachowania |
| Broadcast | Block hash = null | OK |
| Broadcast | Wysłanie bloku jako transakcji | OK, ale rozesłany dalej, chociaż nikt nie wykopał z powodu błędów weryfikacji |
| Broadcast | Prev\_hash = null i difficulty 0 | OK, ale zaczął próbować synchronizować łańcuchy – nie wykonano, bo nikt nie znał bloku null |
| Broadcast | Brak danych wewnętrznych | OK |
| Broadcast | Brak danych | OK, nieprzewidziany wyjątek w konsoli |
| Register\_neighbor | Losowe adresy | OK, ale są dodawane |
| Register\_neighbor | Brak danych wewnętrznych | OK, ale rejestruje undefined |
| Register\_neighbor | Brak danych | Odpowiedź nie w formacie JSON, nieprzewidziany wyjątek w konsoli |
| Join\_network | Odpytanie pierwszego węzła INIT | Nieskończona odpowiedź, ale węzeł przetwarza bloki i działa dalej |
| Leave\_network | Odpytanie węzła INIT | OK, ale nie w formacie json |
| neighbors | DELETE | OK, ale nie w formacie json |
| Neighbors | PUT | OK, ale przyjmuje wszystkie wartości i je zapisuje (nawet integery, puste obiekty) |
| Parent | Niepoprawne hashe | OK |
| Broadcast | Pusty sync\_chain | Nie w formacie json, status 500, nieprzewidziany wyjątek w konsoli |

Podsumowując zebrane dane – implementacja węzła pod kątem odporności na nietypowe/błędne żądania **nie może zostać uznana z pewnością za bezpieczną** mimo, że **nie udało się jej zepsuć**. Powodem jest wiele przesłanek **niedokładnej implementacji pod kątem sprawdzania danych** wejściowych. Dla wielu sprawdzonych scenariuszy odpowiedzi zwracane były w złym formacie, bądź jak w przypadku /neighbors dane wejściowe nie były walidowane (!). Z drugiej strony, mechanizmy walidacji w ważniejszych częściach systemu naprawiały błędy popełnione przez prostsze endpointy, więc **nie ma też jasnego dowodu że system jest niebezpieczny**. Jednak zgodnie z zasadami cyberbezpieczeństwa można powiedzieć, że **brak** **dowodu bezpieczeństwa można interpretować jako przesłankę istnienia podatności.**

### Propozycja poprawek

Poprawki oczywiste: lepsze walidowanie danych, większe przyłożenie się do przetwarzania danych i odpowiedzi nawet w mniej istotnych dla systemu endpointach – wkońcu podatność w nieistotnej części zazwyczaj jej miejscem uzyskania dostępu skąd atakujący przechodzi do tych ważniejszych.

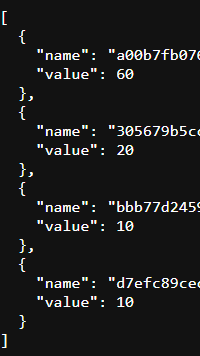
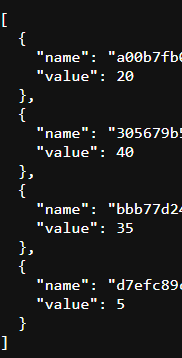
## Złamanie algorytmów kryptograficznych

### Złamanie hasha

Złamanie hasha można zinterpretować jako możliwość powrotu do wartości oryginalnej bądź uzyskanie możliwości wyszukiwania kolizji w krótkim czasie. W projekcie hash jest używany tylko w celach weryfikacji oraz skracania danych (id jako hash z klucza), dlatego pierwsza możliwość nie jest tutaj zagrożeniem. Skuteczny atak na hash dający możliwość łatwego tworzenia kolizji można zasymulować poprzez wyłączenie mechanizmów weryfikujących, ponieważ nie są już wiarygodne. Struktura testu:

* Kopacze na adresach 5000 i 5004
* Nieuczciwy węzeł 5001 – fałszuje zarówno otrzymane transakcje, jak i bloki poprzez podwajanie wartości transakcji
* Wysyłane po kolei transakcje:
  + 10 coinów od 5001 do 5001 (adres 305…)
  + 1 coin od 5003 do 5000 (adres d7ef…) – niepoprawna, bo 5003 nie otrzymał żadnych środków startowych od coinbase
  + 10 coinów od 5001 do 5003 (adres bbb…)
  + 5 coinów od 5003 do 5000 (adres d7ef) – tym razem poprawna
* W idealnych warunkach transakcje powinny zakończyć się stanem:
  + Adres 305… : 10
  + bbb… : 5
  + d7ef… : 5
  + a00b… (konto coinbase): 80

Po przeprowadzeniu testu uzyskano następujące wyniki:

  
Stany kont według węzła 5003 Stany kont według 5000

Pierwszą obserwacją jest to, że fałszerstwo zostało zaakceptowane przez sieć! Jest jednak jeszcze inne interesujące zjawisko, czyli różnica w stanach kont. Jest to spowodowane tym, że transakcja o wartości 5 dla konta d7efc.. pochodzi z węzła 5003 i została wykopana przez węzeł 5004. Skutkuje to tym, że wykopany dla niej blok dociera do 5003 w niezakłóconym stanie, jednak do węzła 5000 dociera przechodząc przez fałszywy węzeł 5001, który podwaja wartość transferu.

Innym zjawiskiem są początkowe transakcje pochodzące z 5001. Oryginalnie miały one wartość 10, ale z perspektywy 5000 są warte 20 (transakcja podwojona przez fałszera), natomiast z widoku 5003 warte są aż 40 (najpierw transakcja podwojona, a później blok wykopany przez 5000 podwojony w drodze do 5003).

Trzeba się jednak zastanowić jak wyglądało by tworzenie kolizji? Żeby otrzymać kolizję trzeba zmodyfikować jakieś pole – w przypadku transakcji są to typ transakcji, adresy kont, wartość oraz timestamp. Węzeł atakujący musiałby więc manipulować timestamp-em, by otrzymać kolizję, ponieważ typ, adres wychodzący i adres docelowy mają określony zbiór wartości, natomiast stany kont raczej mają stosunkowo mały przedział dozwolonych wartości spowodowany stanami kont (istnieje jednak szansa, że na wskutek procesów ekonomicznych użytkownicy przechowywali by miliardy monet, co całkowicie odwróciło by sytuację!). Można więc ten atak częściowo skontrować sprawdzaniem znaczników czasowych tak, aby nie akceptować przyszłych, bądź zbyt starych transakcji – wtedy ryzyko, że kolizja wystąpi dla pobliskiego znacznika jest bardzo niskie.

W przypadku bloków węzeł atakujący ma dowolność modyfikacji wszystkich pól oprócz wnętrza transakcji, ponieważ kolizje może znaleźć z użyciem nonce, który nie jest sprawdzany. W kontekście podmiany wartości bądź adresu transakcji nie daje to żadnych możliwości, bo hash transakcji jest weryfikowany również oddzielnie od hasha bloku.

Podsumowując, złamanie hasha polegające na ułatwieniu wyszukiwania kolizji **niszczy bezpieczeństwo i potencjalnie spójność** całej sieci w aktualnej implementacji.

### Złamanie RSA

Ponieważ wykorzystywany jest asymetryczny algorytm RSA to największym zagrożeniem prawdopodobnie jest uzyskanie dostępu do klucza prywatnego na bazie publicznego np. na skutek dynamicznego rozwoju komputerów kwantowych. Na taki atak nie ma zabezpieczeń – kiedy do niego dojdzie to sieć przestanie działać, ponieważ pozwoli atakującemu na wysyłanie sobie dowolnych ilości monet i podpisywanie ich kluczami innych osób. Sprawa jest tak prosta, że nie potrzeba żadnych testów.

#### Ryzyko ślepego podpisu

Teoretycznie istnieje ryzyko wykonania przez użytkownika ślepego podpisu, który atakujący mógłby wykorzystać do wyciągnięcia klucza prywatnego. Scenariusz to sytuacja, w której wiarygodny użytkownik chce zapłacić za coś złodziejowi (nie wie, że to złodziej – może np. płacić za jakąś prawdziwą usługę). Wtedy złodziej preparuje swój adres, by wykonać atak na ślepy podpis (ślepota polega na tym, że nikt naprawdę nie wie czy docelowy adres konta to nie jest coś innego) nawet jeśli oznaczało by to utratę paru monet – potencjalny zysk znacznie większy.

Takie ryzyko jest jednak w aktualnej implementacji stosunkowo niskie, ponieważ transakcja posiada znacznik czasowy ustalany przez wysyłającego, czyli podpisującego – oznacza to, że atakujący musiałby przewidzieć jego wartość z ogromną dokładnością. Sprawia to, że taki atak nie jest realnym zagrożeniem, ale jest wykonywalny, czyli gdyby atakujący sprzedawał tysiące usług na minutę to w końcu mógłby trafić. W implementacji nie jest użyty żaden mechanizm zabezpieczający przed tym, więc **istnieje małe, ale ryzyko naruszenia bezpieczeństwa**!

### Propozycje poprawek

Dodanie weryfikacji znacznika czasowego tak, aby uczciwe węzły akceptowały tylko niedawne, przeszłe transakcje bardzo mocno ograniczyło by ryzyko akceptacji fałszywej transakcji, ponieważ to właśnie timestamp jest jedynym modyfikowalnym polem transakcji. Jak już zostało wspomniane, gdyby w skutek procesów ekonomicznych doszło do sytuacji gdzie przeciętny użytkownik ma miliony monet to pole amount stało by się dodatkową opcją do modyfikacji w poszukiwaniu kolizji znacznie zwiększając podatność sieci. Oznacza to, że nawet **taka poprawka tylko zmniejsza, ale nie eliminuje negatywnych skutków** sytuacji, czyli **nie daje bezpieczeństwa**.

## Wysyłanie przestarzałych bloków