SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 678

SIGURNOSNE RANJIVOSTI U PAMETNIM UGOVORIMA

Lovre Mitrović

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

ZAVRŠNI RAD br. 678

SIGURNOSNE RANJIVOSTI U PAMETNIM UGOVORIMA

Lovre Mitrović

Zagreb, lipanj 2022.

SVEUČILIŠTE U ZAGREBU
FAKULTET ELEKTROTEHNIKE I RAČUNARSTVA

Zagreb, 11. ožujka 2022.

ZAVRŠNI ZADATAK br. 678

Pristupnik:	**Lovre Mitrović (0036523529)**
Studij:	Elektrotehnika i informacijska tehnologija i Računarstvo
Modul:	Računarstvo
Mentor:	doc. dr. sc. Ante Đerek

Zadatak: **Sigurnosne ranjivosti u pametnim ugovorima**

Opis zadatka:

Pametni ugovori su aplikacije koje se na pouzdan i deterministički način izvršavaju na kriptografskom lancu blokova koristeći virtualni stroj i protokol raspodijeljenog konsenzusa. Te aplikacije mogu implementirati autonomnu poslovnu logiku te raspolagati kriptovalutama. Stoga je od iznimne važnosti osigurati da su pametni ugovori otporni na napade i zlouporabu. U sklopu završnog rada potrebno je istražiti sigurnosna svojstva koja pružaju pametni ugovori na nekoj često korištenoj platformi te česte ranjivosti u implementaciji pametnih ugovora. U praktičnom dijelu rada, potrebno je ili osmisliti i ostvariti sustav koji će demonstrirati neželjene interakcije i iskorištavanje ranjivosti u pametnim ugovorima ili osmisliti i ostvariti jednostavnu metodu provjere je li pametni ugovor otporan na neku konkretnu ranjivost. Radu je potrebno priložiti izvorni kod razvijenih i korištenih programa, citirati korištenu literaturu i navesti dobivenu pomoć.

Rok za predaju rada: 10. lipnja 2022.

Zahvaljujem se

svom mentoru Anti Đereku koji je usmjeravao ovaj rad od misli do realizacije i bio na raspolaganju sa svojim ogromnim poznavanjem ove fascinirajuće tematike

SADRŽAJ

1. Uvod

Prva praktična primjena tehnologije lanca blokova je Bitcoin koji služi kao decentralizirani elektronički novac. Od pojave Bitcoina 2009. godine[7] tehnologija lanca blokova je ubrzano napredovala i postala prihvaćena i korištena u mnogim industrijama. Bitcoin je prihvaćen do te razine da je legalno sredstvo plaćanja u El Salvadoru od 9. lipnja 2021. godine.[11]

Sljedeći značajni korak u razvoju lanca blokova je pojava Ethereuma 2015. godine [8]. Najvažniji iskorak Ethereuma je postavljanje poslovne logike na lanac blokova u obliku programa koje nazivamo **pametni ugovori** ,a ne samo transakcija. Ako Bitcoin uspoređujemo s tablicom, onda je Ethereum treba usporediti s potpuno funkcionalnim računalom. Kriptovaluta koju Ethereum koristi za transakcije je Ether.

Kao i u razvoju "normalni" programa niti pametni ugovori nisu otporni na razne ranjivosti. Ali zbog prirode lanca blokova jednom napravljen ugovor više nije moguće izmjenjivati također bajtkod ugovora je javno vidljiv na lancu blokova. Kad uz sve to pridodamo činjenicu da ugovori barataju kriptovalutama dolazimo do zaključka da jednostavna ranjivost u ugovoru može prouzročiti velike monetarne posljedice. Stoga je jako bitno da ugovor ne sadržava sigurnosne ranjivosti. U nastavku rada razmotrit ćemo nekoliko odabranih ranjivosti te njihove uzroke i preventivne tehnike. Također na kraju ću predložiti rješenje koje u ugovoru prepoznaje sigurnosni propust **iluzije entropije**. Takav propust se najčešće događa u ugovorima koji imaju potrebu za generiranjem slučajnih brojeva poput ugovora koji bi implementirao kockarnicu.

U svrhu izrade rješenja koje prepoznaje ranjivost koristiti ćemo metodu **statičke analize koda**. Rješenje ćemo testirati na nekoliko pokaznih primjera. Za kraj slijedi diskusija o navedenom rješenju, njegovim propustima i prostoru za poboljšanje te detaljnije razmatranje tehnika prevencije navedene ranjivosti u implementaciji pametnih ugovora.

Poglavlje 2 daje uvid u tehnološku pozadinu korištenih tehnologija. U poglavlju 3 razmatramo odabrane ranjivosti i njihove preventivne tehnike. U poglavlju 4 razmatramo sustav za detekciju ranjivosti iluzije entropije.

2. Tehnološka pozadina

2.1. Lanac blokova

Lanac blokova je raspoređena baza podataka koja za razliku od tradicionalnih baza podataka informacije grupira u blokove ograničenog kapaciteta. Kad je blok popunjen on se povezuje s prethodnim blokom te tako povezani blokovi čine lanac blokova. Svaki blok uz podatke koje želimo pohraniti sadržava svoj sažetak i sažetak prethodnog bloka. [9] Da bi izmijenili podatak u nekom prethodnom bloku potrebno je izmijeniti i sve sljedeće blokove u lancu. To svojstvo zajedno s činjenicom da je lanac blokova raspoređen u praksi rezultira da su podaci koji su jednom zapisani nisu izmjenjivi.

U ovakvu bazu podataka se mogu pohranjivati različite vrste informacije poput novčanih transakcija, praćenja vlasništva, medicinskih podataka, ugovora i slično. Prva praktična uporaba lanca blokova, koja je u funkciji od 2009. godine, je Bitcoin koji je zamišljen kao alternativa novcu u elektroničkom istorazinskom obliku *(eng. peer-to-peer)*. [10]

2.2. Ethereum

Sljedeća stepenica u evoluciji primjene tehnologije lanca blokova nakon Bitcoina je Ethereum platforma. Ethereum podiže lanac blokova s razine rapoređene baze podataka na razinu raspoređenog globalnog računala. Ethereum protokol postoji u svrhu održavanja kontinuiranih, neprekinutih i neizmjenjivih operacija nad stanjem stanjima tog računala [1]. Za bilo koji blok Ethereum ima točno jedno stanje i virtualni stroj Ethereuma (EVM) je ono što definira pravila za izračunavanje sljedećeg stanja iz bloka u blok [1].

Ethereum je ovim uveo pametne ugovore koji su u suštini programi koji se izvode na virtualnom stroju. Iz svojstava lanca blokova slijedi da su takvi programi, jednom kad su postavljeni na lanac, trajni, deterministički i neizmjenjivi. Iz aspekta razvoja

ovakvih programa to nije povoljno jer ne ostavlja prostora za ispravljanje pogrešaka što uključuje i one sigurnosne koje mogu dovesti do velikih posljedica poput onih financijskih i slično. Usprkos tome pametni ugovor su ponajviše pronašli svoju primjenu u financijskom sektoru decentraliziranih financija (DeFi) i decentraliziranih aplikacija.

2.2.1. Solidity

Možemo nadograditi analogiju između virtualnog stroja Ethereuma i računala to jest CPU-a uspoređujući strojni kod CPU-a s bajt kodom virtualnog stroja. Pametne ugovore je moguće pisati u bajt kodu isto kao što je programe za klasična računala moguće pisati u strojnom kodu[4], ali zbog ograničenosti resursa kao i veće mogućnosti pogrešaka izbjegavamo takvu praksu i služimo se višim programskim jezicima. Najpoznatiji takav viši programski jezik je Solidity koji bi u našoj analogiji odgovarao programskom jeziku Java.

2.3. Statička analiza koda

Statistička analiza koda je metoda pronalaženja pogrešaka programa koja se izvodi pregledom koda bez njegovog pokretanja za razliku od dinamičke analize koda koje pokreće program. Tehnike koje se koriste u statičkoj analize koda proizlaze iz uglavnom iz tehnologije koju koristi jezični procesor. [2] Primjeri tih tehnika su analiza toka podataka, graf kontrolnog toka, analiza mrlja i leksička analiza. [2] Jedan od poznatijih alata za statističku analizu koda u Solidity jeziku koji se koristi u ovom radu je radni okvir Slither napisan u Pythonu 3.

3. Odabrane sigurnosne ranjivosti

Kao što je već napomenuto jednom kad je pametni ugovor postavljen na lanac blokova više nije izmjenjiv. Iz tog razloga je izuzetno bitno da ne sadržava sigurnosne ranjivosti. Dobra praksa je defenzivno programiranja koje naglašava jednostavnost i minimalizam koda, ponovno korištenje već napisanog koda poput uporabe postojećih i provjerenih biblioteka, pisanje razumljivog i kvalitetnog koda te veliku pokrivenost testiranja. [4]

3.1. Ponovni ulazak

Pametni ugovori imaju mogućnost pozivati druge pametne ugovore što omogućava ponovnu uporabu već napisanog koda poput biblioteka. Također ugovori obično obavljaju i transakcije s tokenom Ether te imaju mogućnost implementiranja *fallback* funkcije koja se poziva kad ugovor primi Ether. [4]

3.1.1. Ranjivost

Sigurnosna prijetnja je trenutak kad ranjivi ugovor šalje Ether na nepoznatu adresu. Napadač na toj vanjskoj nepoznatoj adresi može konstruirati ugovor koji ima implementiranu *fallback* funkciju sa zloćudnim kodom. Kada ranjivi ugovor pošalje Ether poziva se navedena *fallback* funkcija koja opet može pozvati funkciju na ranjivom ugovoru. Od tud termin “ponovni ulazak”. [4]

3.1.2. Preventivne tehnike

Postoje broje tehnike koje pomažu smanjiti mogućnost ovakve ranjivosti.

Prva je korištenje ugrađene *transfer* funkcije koja koristi samo 2300 jedinice goriva prilikom poziva vanjskog ugovora što nije dovoljno da vanjski ugovor obavlja daljnje pozive što sprječava pozivanje ranjivog ugovora.

Druga preventivna tehnika je da sva logika promjene stanja varijabli na ugovoru se izvrši prije slanja Ethera na vanjsku adresu.

I treća je korištenje mutex objekta to jest varijable koja zaključava ugovor dok se kod izvršava te tako onemogućuje ponovni ulazak tijekom izvršavanja.

3.2. Neočekivani Ether

Uobičajeno je da kad pametni ugovor prima Ether izvršava kod funkcije koju je specificirao pošiljatelj ili u slučaju da funkcija nije specificirana *fallback* funkcije. Međutim postoje dva načina kako poslati Ether na adresu bez da se izvrši kod. Što ako programer nije bio dovoljno pažljiv može uzrokovati daljnje probleme. [4]

3.2.1. Ranjivost

Ranjivost proizlazi iz loših početnih pretpostavki programera koji nije svjestan da postoje načini dolaska Ethera na adresu ne koristeći *payable* funkcije što dovodi do loših pretpostavki o količini Ethera na adresi. Navedene loše pretpostavke mogu prouzročiti niz ranjivosti na primjer lošim korištenjem *this.balance*.

Prvi način slanja Ethera bez aktivacije koda je koristeći *self-destruct* funkciju. Svaki pametni ugovor ima mogućnost implementacije *self-destruct* funkcije koja kad se pozove uklanja kod s adrese ugovora i šalje sav Ether na adresu koja je zadana kao argument funkcije. U ovom slučaju napadač konstruira novi ugovor sa *self-destruct* funkcijom koju poziva predajući adresu ranjivog ugovora kao argument. Naravno prije pozivanja *self-destruct* funkcije potrebno je poslati željenu količinu Ethera na upravo konstruirani ugovor. Taj Ether će potom biti proslijeđen na ranjivi ugovor prilikom poziva *self-destruct* funkcije.

Drugi način proizlazi iz činjenice da je cijeli lanac blokova deterministički sustav. Te je moguće izračunati adresu na koju će se nalaziti neki ugovor prije nego što je postavljen na lanac. Napadač može poslati Ether na adresu na kojoj se još ne nalazi ništa te tako potencijalno prouzročiti probleme ugovoru koji će biti postavljen na tu adresu.

3.2.2. Preventivne tehnike

Treba izbjegavati ovisnost logike pametnog ugovora o količini Ethera koji se nalazi na njemu. Ako takvu ovisnost nije moguće ukloniti treba uvesti varijablu koja će pratiti

stanje očekivanog Ethera i koristiti takvu varijablu umjesto *this.balance*. [4]

3.3. Iluzija entropije

Kao što je već navedeno lanac blokova je deterministički sustav ,a na takvom sustavu nema nesigurnosti sve što se događa je izračunljivo. Poznavajući trenutno stanje, zadane akcije i pravila moguće je izračunati sljedeće stanje.

3.3.1. Ranjivost

U ugovorima koji su implementirani u svrhu kockanja gdje se ugovor ponaša kao kuća dolazimo do potrebe za slučajnošću. Primjerice generator slučajnih brojeva koji će simulirati rulet. Često bi programer koji implementira ovo svojstvo posegao za varijablama lanca blokova poput trenutnog hasha, timestampa, težine (eng. difficulty) kao izvorom entropije. Te nad tim varijablama izvršio funkciju sažetka (eng. hash). Problem je što su funkcije sažetka determinističke i predvidljive ,a varijable lanca podložne manipulaciji od strane rudara koji izrudari blok. Uglavnom su ovakvi napadi neisplativi napadaču ,ali svakako treba biti svjestan potencijalnih ranjivosti.

3.3.2. Preventivne tehnike

Preporučuje se izbjegavati koristiti lanac blokova kao izvor entropije. Ako implementiramo sustav gdje je potrebna entropija između korisnika poput kockanja između dva igrača možemo koristiti obrazac *obaveži-pokaži (eng. commit-reveal)* ili se poslužiti sustavom za stvaranje entropije u grupi korisnika poput *RandDAO*. Međutim ako implementiramo sustav između igrača i kuće zadatak stvaranje entropije možemo prebaciti na neki centralizirani entitet izvan lanca blokova. [4]

3.4. Brojevi s pomičnom točkom i preciznost

Kako Solidity ne podržava brojeve s pomičnom točkom *(eng. floating point)* njih će samostalno trebati konstruirati programer preko cijelih brojeva što može dovesti do stvaranja grešaka i sigurnosnih ranjivosti. Potreba za brojevima s pomičnom točkom se javlja na primjer kod ugovora koji vode evidenciju o ekonomskim odnosima na primjer kupovina i prodaja prilagođenim vlastoručno implementiranih tokena.

3.4.1. Ranjivost

Mnogo je stvari koje može poći pogrešno dok samostalno implementiramo brojeve s pomičnom točkom na primjer kod cijelih brojeva ako je djeljenik manji od djelitelja rezultat je automatski nula. Jedan ilustrativni primjer je navedeni ugovor korišten kao pokazni primjer u [4]

```
1  contract FunWithNumbers {
2      uint constant public tokensPerEth = 10;
3      uint constant public weiPerEth = 1e18;
4      mapping(address => uint) public balances;
5
6      function buyTokens() external payable {
7          // convert wei to eth, then multiply by token rate
8          uint tokens = msg.value/weiPerEth*tokensPerEth;
9          balances[msg.sender] += tokens;
10     }
11
12 }
```

U ugovoru kako je gore napisan nije moguće kupiti 1 ili 11 tokena već samo broj tokena koji je višekratnik broja 10. Ranjivost se nalazi u 8 liniji gdje se zbog operacije dijeljenja gubi na preciznosti.

3.4.2. Preventivne tehnike

Potrebno je osigurati da su brojevi u omjerima dovoljno veliki. U našem konkretnom primjeri to bi značilo da umjesto *tokensPerEther* koristimo mjeru *weiPerTokens*. Odnosno u 8 retku u naredbi pridruživanja s desne strane koristimo izraz *msg.value/weiPerTokens* što bi omogućilo dovoljno veliku preciznost za kupovinu minimalno jednog tokena, a ne 10.

Druga metoda je osigurati da su operacije u redoslijedu koji omogućuje veću preciznost na primjer da umjesto *msg.value/weiPerEth*tokenPerEth* koristimo redoslijed *msg.value*tokenPerEth/weiPerEth*. [4]

Treći način da se koriste već napisane i provjerene biblioteke koje omogućavaju rad s brojevima s pomičnom točkom poput ABDKMathQuad.sol [3].

3.5. Umetanje ispred

Priroda lanca blokova je takva da je redoslijed transakcija bitan što otvara mogućnost za ranjivost umetanja ispred.

3.5.1. Ranjivost

Kao što je spomenuto u uvodnom dijelu lanac blokova podatke sprema u blokove međutim ti podaci su spremljeni na lanaca tek kad rudar izrudari blok. Zbog prevelikog broja transakcija rudar mora odlučiti koje će transakcije spremiti u blok. Rudar ih može spremati proizvoljno, ali najčešće na osnovu količine goriva transakcije na način da one s najvećom količinom imaju prednost. Zamislimo ugovor koji daje nagradu igraču koji prvi riješi slagalicu. Napadač može osluškivati transakcije i kada naiđe na transakciju koja uspješno rješava slagalicu prekopira podatke u svoju, ali postavi značajno veću količinu goriva tako da se njegova transakcije prije obradi. Napadač može biti i sam rudar koji proizvoljno stavi svoju transakciju u blok ,a izostavi onu originalnu. Vjerojatnost da je napadač rudar je značajno manja jer je vjerojatnost rješavanja i rudarenja bloka za pojedinačnog rudara izuzetno mala.

3.5.2. Preventivne tehnike

U prethodnom primjeru zaštita od napadača koji nije rudar je postavljanje gornje granice na količinu goriva što će onemogućiti da napadač postavi količinu goriva transakcije veću od originalne ako je originalna već postavljena na najveću moguću dopuštenu količinu.

Otpornija metoda bi bila korištenje obrasca *obaveži-pokaži*. Ovakva metoda prevenira napad čak i ako je napadač sam rudar. [4]

4. Sustav za detekciju ranjivosti iluzije entropije

4.1. Modeliranje problema

4.1.1. Slither

Slither je radni okvir otvorenog koda za statičku analizu Solidity programskog jezika. Slither analizira ugovor koristeći statičku analizu kroz nekoliko koraka. Kao inicijalni ulaz Sliter koristi apstraktno sintaksno stablo kojeg generira Solidity prevoditelj iz izvornog koda. U prvoj fazi Slither izvlači bitne informacije poput grafa nasljeđivanja, grafa kontrolnog toka i liste izraza. Nadalje Slither pretvara cijeli kod ugovora u SlitherIR kao internu reprezentaciju jezika. Tijekom treće faze, stvarne analize koda, Slither izračunava skup unaprijed definiranih analiza koje pružaju dodatnu informaciju ostalim modulima. [6]

Slither radni okvir je moguće proširiti vlastitim detektorom. U nastavku ćemo razmotriti kako napraviti detektor za detekciju loših praksa prilikom generiranja vlastitih slučajnih brojeva.

4.1.2. Problem

U nastavku ćemo razmotriti ugovor koji za izvor entropije uzima dvije varijable s lanca blokova kao izvor entropije te nad njima izvodi funkciju sažetka. Pošto se primjer zamišljen da vrati nasumičan broj od 0 do 99 broj dobiven funkcijom sažetka nad izvorom entropije ćemo podijeliti sa 100 i uzeti njegov ostatak. U programerskoj praksi nasumičan broj se najčešće ne generira u funkciji u kojoj se koristi već se često napravi zasebna funkcija koja generira nasumičan broj te se koristi njen rezultat. Stoga ćemo ugnijezditi funkciju *random* koja generira slučajni broj u funkciju *play* koja će biti izložena za javno pozivanje.

```
contract Random {

    function random() private view returns(uint){
        uint source = block.difficulty + block.timestamp;
        return uint(keccak256( abi.encodePacked(source)) ) % 100;
    }

    function play() public returns(uint){
        return random();

    }

}
```

4.1.3. Pretpostavke ranjivosti

U nastavku slijede pretpostavke koje trebaju biti zadovoljene da bi klasificirali funkciju kao potencijalno ranjiv generator slučajnih brojeva:

1. Funkcija čita varijable stanja lanca blokova
2. Funkcija koristi funkciju sažetka
3. Funkcija kao rezultat vraća izraz oblika *hash(src)* gdje je *hash* funkcija sažetka, a *src* varijabla stanja lanca blokova ili varijabla ovisna o nekoj varijabli stanja lanca blokova

Treća pretpostavka obuhvaća prve dvije, ali sam je uključio radi brže implementacije. Na primjer ako već uočimo da se nigdje ne koristi funkcija sažetka ili ne čita varijabla lanca blokova nema potrebe provjeravati je li rezultat koji se vraća sadržava izraz sadržan u trećoj pretpostavci.

4.2. Implementacija

4.2.1. Početni koraci

Novi detektor ćemo implementirati u Python 3.7 programskom jeziku unutar Slither radnog okvira. Detektor će nasljeđivati apstraktnu klasu *AbstractDetector*. Za početak potrebno je nadjačavati postojeće varijable da bi omogućili korištenje alata iz komandne linije (ARGUMENT, HELP, IMPACT, CONFIDENCE) i varijabli za wiki stra-

nicu (WIKI, WIKI_TITLE, WIKI_DESCRIPTION, WIKI_EXPLOIT_SCENARIO, WIKI_RECOMMENDATION).

```
class RandomDetector(AbstractDetector):
    # slither /path/tocontrct --detect random-detector
    ARGUMENT = 'random'
    HELP = 'Function which produces bad random numbers'
    IMPACT = DetectorClassification.HIGH
    CONFIDENCE = DetectorClassification.LOW

    WIKI = 'No wiki'

    WIKI_TITLE = 'Bad random number generator'
    WIKI_DESCRIPTION = 'Detects function which generates bad random
        number'
    WIKI_EXPLOIT_SCENARIO = 'Blockchain variables are deterministic
        and predictable'
    WIKI_RECOMMENDATION = 'Do not generate random numbers from
        blockchain varialbes'
```

Logika detektora se stavlja u metodu koja nadjačava *_detect* metodu. Po uzoru na [5] odgovornosti te metode ćemo podijeliti na tri dijela:

1. Prolazak kroz sve ugovore zadane u komandnoj liniji (metoda *_detect*)
2. Za svaki ugovor prolazak kroz sve funkcije koje taj ugovor deklarira (metoda *detect_random*)
3. Provjera je li funkcija zadovoljava pretpostavke ranjivosti (metoda *random*)

4.2.2. Prolazak kroz ugovore

Funkcija *_detect* prolazi kroz sve zadane ugovore i nad svakim poziva metodu *detect_random* koja vraća listu funkcija i čvorova koji su označeni kao potencijalno ranjivi i generira pripadajuću analizu.

```
    def _detect(self):

        results = []

        info = []
        for c in self.contracts:
            detected_random = self.detect_random(c)
            for (func, nodes) in detected_random:
```

```
                info = [func, " generates bad random numbers\n"]
                info += ['\tDangerous calls:\n']

                for node in nodes:
                    info += ['\t\t ', node, '\n']

            res = self.generate_result(info)
            results.append(res)

        return results
```

4.2.3. Prolazak kroz funkcije

Metoda *detect_random* će za zadani ugovor proći kroz sve funkcije koje je taj ugovor deklarirao. Za svaku takvu funkciju nad njom poziva metodu *random* koja vraća listu ranjivih čvorova u toj funkciji. Također prolazi i kroz sve interne pozive te funkcije (što rješava naš problem ugniježđenih funkcija opisan u 4.1.2) i nad njima poziva metodu *random* te sprema i njihove ranjive čvorove.

```
    def detect_random(self, contract: impContract):

        ret = []

        for f in [f for f in contract.functions if f.
            contract_declarer == contract]:
            nodes = self.random(f)
            internal_fn_calls = [call for call in f.
                all_internal_calls() if call in contract.functions]
            for internal_fn in internal_fn_calls:
                nodes += self.random(internal_fn)
            if nodes:
                ret.append((f, nodes))

        return ret
```

4.2.4. Detektiranje ranjive funkcije

Metoda *random* je središnji našeg detektora koji označava funkciju kao potencijalno ranjivu na osnovu pretpostavki iz 4.1.3.

Radni okvir Slither ima ugrađeno prepoznavanje za ono što naziva Solidity varijablama primjerice *block.timestamp*, *msg.sender*, *msg.gas* i slične. Skup tih varijabli

obuhvaća varijable stanja lanca blokova koje su posebno interesantne jer se često koriste kao izvor entropije poput *block.timestamp*. Ostale varijable u tom skupu poput *msg.sender* i *msg.gas* također nisu pogodne kao izvor entropije jer njima manipulira korisnik ,ali ionako se u praksi ni ne koriste za to što nije slučaj kod varijabli stanja lanca blokova. Ako se niti jedna takva Solidity varijabla ne čita, funkciju nećemo označiti kao potencijalno ranjivu.

U sljedećem dijelu ručno su izdvojene funkcije sažetka dostupne u Solidity jeziku u listu *HASH_FUNCTIONS*. Slither ima ugrađenu podršku za prepoznavanje Solidity funkcija koje uključuju i zadane funkcije sažetka ,ali i mnoge druge često korištene funkcije poput *revert()*. Zatim se stvara lista *called_hash_functions* koja sadržava sve funkcije sažetka koje su interno pozivane u našoj funkciji. Ako je ta lista prazna odnosno ako nije bilo poziva funkcija sažetka, funkciju nećemo označiti kao potencijalno ranjivu.

Sada kada smo utvrdili da je došlo do čitanja varijabli lanca blokova i pozivanja funkcija sažetka potrebno je provjeriti je li rezultat funkcije izraz naveden u pretpostavci 3. u 4.1.3. odjeljku. Za početak od svih varijabli koje je funkcija pisala izdvajamo one koje su definirane pomoću varijabli lanca blokova. Zatim se parsira izraz koji funkcija vraća kao rezultat (redci 33-40). Ako je uspješno pronađen uzorak naveden u 4.1.3. podiže se zastavica *vulnerable*. Ako je zastavica podignuta funkcija se označava kao ranjiva i vraćaju se čvorovi funkcije inače se vraća prazna lista, odnosno funkcija se ne označava kao ranjiva.

```
1     @staticmethod
2     def random(func: Function):
3
4         # 1) it doesnt reads from block variables or other
              blockchain variables
5         if len(func.solidity_variables_read) == 0:
6             return []
7
8         # 2) it doesnt call on hash functions
9         HASH_FUNCTIONS = ["keccak256()", "keccak256(bytes)",  #
              Solidity 0.5
10                          "sha256()", "sha256(bytes)",  # Solidity
                                0.5
11                          "sha3()",
12                          "ripemd160()", "ripemd160(bytes)"]  #
                                Solidity 0.5]
13        called_hash_function = []
14        for f in func.internal_calls:
```

```
            if not isinstance(f, SolidityFunction):
                continue
            else:
                if f.name in HASH_FUNCTIONS:
                    called_hash_function.append(f)

        if len(called_hash_function) == 0:
            return []

        # 3.0)finding variables in which block variables are written
        # eg source = block.timestamp
        vars_derived_from_block = [var.name
                                   for var in func.variables_written
                                   for block_var in func.
                                       solidity_variables_read
                                   if block_var.name in str(var.
                                       expression)]

        # 3.1) it doesnt contain expression hash(blockchain variable
            )
        vulnerable = False
        return_expression = str(func.all_expressions()[-1])
        for hash_func in HASH_FUNCTIONS:
            if hash_func in return_expression:
                args = return_expression.split(hash_func)[1].rsplit(
                    ')',1)[0]
                if any(map(lambda block_var:block_var in args,
                           [o.name for o in func.
                               solidity_variables_read] +
                               vars_derived_from_block)):
                    vulnerable = True
                    break

        if not vulnerable: return []

        ret = []
        for node in func.nodes:
            ret.append(node)

        return ret
```

4.3. Testiranje

U ovom poglavlju testirati ćemo funkcionalnost detektora. U prvom dijelu razmatramo funkcije s različitim kombinacijama izvora entropije. A u drugom funkciju koja je delegirala generiranje nasumičnih brojeva ranjivoj funkciji.

4.3.1. Funkcija za nasumičan broj

Razmatram za početak najjednostavniji slučaj gdje pozivam funkciju sažetka direktno nad varijablom stanja lanca blokova.

```
1    function random_simple() private view returns(uint){
2        return uint(keccak256( abi.encodePacked(block.difficulty)) )
             % 100;
3    }
```

Detektor je bez problema pronašao ranjivost u drugoj liniji.

Sada ćemo izvršiti funkciju sažetka nad varijablom koja ovisi o varijabli lanca blokova.

```
1    function random_src() private view returns(uint){
2        uint source = block.difficulty;
3        return uint(keccak256( abi.encodePacked(source)) ) % 100;
4    }
```

Detektor također pronalazi ranjivost. Tu varijablu *source* ćemo sada zakomplicirati tako da ovisi o dvije varijable lanca blokova.

```
1    function random_composed() private view returns(uint){
2        uint source = block.difficulty * block.timestamp;
3        return uint(keccak256( abi.encodePacked(source)) ) % 100;
4    }
```

Detektor također pronalazi ranjivost.

Lažno negativan slučaj

Sada ćemo konstruirati posredničku ovisnost *source* varijable o varijablama lanca blokova preko varijable *temp*.

```
1    function random_complex() private view returns(uint){
2        uint temp = block.difficulty * block.timestamp;
3        uint source = temp;
4        return uint(keccak256( abi.encodePacked(source)) ) % 100;
5    }
```

U ovom slučaju detektor nije pronašao ranjivost. Izgrađenom detektoru nedostaje logika koja provjerava dublje posredničku ovisnost korisničkih varijabli o varijablama lanca blokova. Ovaj slučaj je **lažno negativan** na ranjivost.

4.3.2. Funkcija unutar funkcije

Sada ćemo iskoristi funkciju najveće kompleksnosti iz posljednji primjera koju je detektor označio kao ranjivu i nju pozvati u nekoj drugoj funkciji. Česta programerska praksa je delegiranja zadatka poput generiranja nasumičnog broja nekoj drugoj funkciji pa pozivanje iste kroz više funkcija. Funkcija koja poziva ranjivu funkciju treba biti označena kao ranjiva.

```
1     function random_composed() private view returns(uint){
2         uint source = block.difficulty * block.timestamp;
3         return uint(keccak256( abi.encodePacked(source)) ) % 100;
4     }
5
6         function play() public view returns(uint){
7         return random_composed();
8
9     }
```

Detektor je funkciju *play* označio kao potencijalnu ranjivu. Za zadatak provjere internih poziva zaslužna je metoda *detect_random* opisana u 4.2.3.

Po uzoru na razvoj testnih slučajeva u 4.3.1 konstruirat ćemo testni primjer u kojem je rezultat ranjive funkcije spremljen u varijablu koja će se vratiti kao rezultat.

```
1     function random_composed() private view returns(uint){
2         uint source = block.difficulty * block.timestamp;
3         return uint(keccak256( abi.encodePacked(source)) ) % 100;
4     }
5
6     function play() public view returns(uint){
7         uint temp = random_composed();
8         return temp;
9
10    }
```

Detektor je pronašao ranjivost. Posrednička varijabla u ovom slučaju ne predstavlja problem. Jer je samo korištenje ranjive funkcije dovoljno da kompromitira cijelu funkciju u kojoj je pozvana. Za ovakvu funkcionalnost je također zadužena metoda *detect_random*.

Lažno pozitivan slučaj

Potaknut prethodnim primjerom konstruirat ćemo slučaj u kojem se ranjiva funkcija samo poziva ,ali nigdje ne sprema niti koristi.

```
1     function random_composed() private view returns(uint){
2         uint source = block.difficulty * block.timestamp;
3         return uint(keccak256( abi.encodePacked(source)) ) % 100;
4     }
5
6     function play() public view returns(uint){
7         random_composed();
8         return 3;
9
10    }
```

Ovakav slučaj će detektor označiti kao ranjiv iako pozivanje ranjive funkcije nema nikakav utjecaj na ostatak. Ovaj slučaj je **lažno pozitivan** na ranjivost.

4.4. Diskusija

4.4.1. Varijable ovisne o varijablama lanca blokova

Kao što je pokazano u 4.3.1 detektor je sposoban pronaći izraze u kojima se kroz funkciju sažetka provlači varijable lanca blokova ili varijabla direktno ovisna o varijablama lanca blokova. Međutim ako zakompliciramo ovaj slučaj tako da se kroz funkciju sažetka provlači varijabla koja je o varijablama lanca blokova ovisna posredno preko još jedne varijable tada detektor neće moći detektirati ranjivost. Primjer ovakvog propusta ilustriran je u 4.3.1.

Detektor je potrebno nadograditi s dodatnom funkcionalnošću u metodu *random* (prikazanom u 4.2.4) tako da može popratiti međuovisnosti varijabli. Predlažem da metoda, kada ustanovi da su zadovoljene prve dvije pretpostavke, u memoriji izgradi strukturu u obliku stabla za svaku varijablu definiranu od strane korisnika koja se koristi u toj metodi. U korijenu bi se nalazila varijabla koju promatramo ,a djeca bi bila sve one varijable koje su se koristile s desne strane prilikom definicije. Djecu bi proširivali na isti način sve dok ne dođemo do konstanti ili varijabli lanca blokova. Zatim bi za svaku varijablu koja se nalazi u izrazu koji se provlači kroz funkciju sažetka provjerili njezino stablo. Ako to stablo sadržava barem jedan list koji sadrži varijablu lanca blokova zaključili bi da je ta varijabla ovisna o varijabli lanca blokova.

4.4.2. Parsiranje

Kad promatramo izraze koji se provlače kroz funkcije sažetka postavlja se pitanje parsiranja tih izraza. Tijekom faze testiranja nisam naišao na probleme s parsiranjem u testnim primjerima ,ali smatram da bi se ono moglo napraviti na elegantniji i sofisticiraniji način nego što je to napravljeno u razmatranom detektoru.

4.4.3. Isprazno korištenje

Osvrnuo bih se na testni primjer iz 4.3.2 koji je lažno pozitivan. Označavanje ovog primjera kao ranjivog ne predstavlja značajan problem jer malo vjerojatno da će se ikad pojaviti u praksi. Nema smisla pozivati funkciju koja ne mijenja nikakvo stanje već samo vraća rezultat te onda taj isti rezultat nigdje ne iskoristiti.

4.4.4. Vjerojatnost napada

Vjerojatnost iskorištavanja ranjivosti iluzije entropije je mala jer varijablama lanca blokova teško manipulirati ,ali ne i nemoguće stoga bi programer u najmanju ruku trebao biti svjestan ranjivosti koje ostavlja u ugovoru. Na primjer ako rudar želi da *block.timestamp* završava s nulom. On može odlučiti rudariti blokove i ne objavljivati ih na lanac blokova sve dok ne dobije rezultat s kojim je zadovoljan. Zbog sveg navedenog preporučuje se koristiti neki od alternativnih izvora entropije koje ćemo razmotriti u 4.5.

4.5. Alternativni izvori entropije

4.5.1. Obaveži-pokaži obrazac

Navedeni obrazac je već spomenut kao rješenje za neke ranjivosti. Zamišljen je da se provede u dvije faze. U prvoj *"obaveži"* fazi korisnik na lanac blokova objavljuje sažetak tajnog podatka dok u drugoj *"pokaži"* fazi korisnik objavljuje stvarne podatke. Ostali korisnici jednostavno mogu provjeriti podudaraju li se sažetak iz prve faze i podaci iz druge faze te se tako uvjeriti u autentičnost.

4.5.2. RandDAO

RandDAO je primjer kako iskoristi prethodno opisan obrazac u praksi na velikoj skali. Moguće je delegirati zadatak generiranja nasumičnog broja grupi korisnika na lancu

blokova primjer takvog načina je RandDAO. Proces generiranja nasumičnog broja odvija se u tri faze.[12]

U prvoj fazi svi korisnici koji žele sudjelovati u izgradnji nasumičnog broja na RandDAO ugovor šalju Ether kao jamstvo i $sha3(s)$ gdje je s tajni broj koji je odabrao sudionik[12].

U drugoj fazi sudionici šalju svoj tajni broj s koji se onda ispituje podudara li se s prethodno poslanim sažetkom.

U trećoj fazi se svi sakupljeni brojevi provlače kroz funkciju $f(s_1, s_2, ..., s_n)$ čiji rezultat zapisuje u pohranu *(eng. storage)* tog RandDAO ugovora. Zatim svim ugovorima koji su poslali zahtjev pošalje nasumičan broj te sudionicima vrati jamstvo zajedno s profitom podijeljenim jednake dijelove između sudionika. Profit dolazi od naknada koje plaćaju ugovori koji su poslali zahtjev za nasumičnim brojem.

4.5.3. Vanjski centralizirani entitet

Moguće je vrlo jednostavno delegirati zadatak generiranja nasumičnog broja na vanjski entitet koji se ne nalazi na lancu blokova za razliku od prethodne metode gdje se cijeli proces događa na lancu blokova. Na primjer koristeći neki od ponuđenih api servisa za generiranje nasumičnog broja. Kad potvrdimo da podatak stvarno dolazi s tog servisa prenesemo taj podatak na lanac blokova. Ovaj pristup pomiče povjerenje s lanca blokova u ruke trećeg servisa.

5. Zaključak

Kroz ovaj rad ukazao sam na nekolicinu odabranih ranjivosti koje su prisutne u pametnim ugovorima koji se izgrađuju za Ethereum lanac blokova. Prošao sam kroz njihove uzroke i tehnike prevencije koje programer treba implementirati dok izgrađuje svoje pametne ugovore. Zbog inherentnih svojstava lanca blokova jednom objavljen ugovor više nije moguće izmjenjivati stoga je izuzetno bitno da ne sadrži sigurnosne ranjivosti.

Posebnu pažnju posvetio sam ranjivosti iluzije entropije. Zbog deterministočnosti lanca blokova izgradnja nasumičnog broja nije trivijalna. Pokazao sam jednu moguću implementaciju detektora za ovu ranjivost u radnom okviru Slither koji koristi statičku analizu koda. Detektor nije savršen i postoji prostora za poboljšanja što sam detaljnije opisao u poglavlju 4.4. Najznačajniji prostor za nadogradnju je dublja provjera ovisnosti korisničkih varijabli o varijablama lanca blokova. Detektor je napravljen na osnovu pretpostavki opisanih u 4.1.3.

Preporuča se izbjegavanje generiranja nasumičnih brojeva koji kao izvor entropije koriste varijable lanca blokova. Umjesto toga potrebno je koristiti entropiju iz grupe sudionika kao što to radi RandDAO (opisano u 4.5.2) ako želimo da se cijeli proces odvija na lancu blokova. Ili koristiti entropiju iz nekog vanjskog centraliziranog izvora kojem vjerujemo (opisano u 4.5.3) ako želimo dio procesa obaviti izvan lanca blokova.

LITERATURA

[1] Ethereum virtual machine (evm). URL https://ethereum.org/en/developers/docs/evm/.

[2] Static code analysis. URL https://owasp.org/www-community/controls/Static_Code_Analysis.

[3] Abdk-Consulting. Abdk-libraries-solidity/abdkmathquad.sol at v3.0 · abdk-consulting/abdk-libraries-solidity, Mar 2021. URL https://github.com/abdk-consulting/abdk-libraries-solidity/blob/v3.0/ABDKMathQuad.sol.

[4] A.M. Antonopoulos, G. Wood, i G. Wood. *Mastering Ethereum: Building Smart Contracts and DApps*. O'Reilly Media, Incorporated, 2018. ISBN 9781491971949. URL https://books.google.hr/books?id=SedSMQAACAAJ.

[5] Blaž Bagić. Static analysis of ethereum smart contracts. Magistarski rad, University of Zagreb, 2003.

[6] Josselin Feist, Gustavo Grieco, i Alex Groce. Slither: A static analysis framework for smart contracts. Technical report, Trail of Bits, 2019. URL https://www.researchgate.net/profile/Josselin-Feist/publication/335441825_Slither_A_Static_Analysis_Framework_For_Smart_Contracts/links/5d6e731aa6fdccf93d38161c/Slither-A-Static-Analysis-Framework-For-Smart-Contracts.pdf.

[7] Jake Frankenfield. The basics on bitcoin (btc), May 2022. URL https://www.investopedia.com/terms/b/bitcoin.asp.

[8] Jake Frankenfield. What is ethereum?, May 2022. URL https://www.investopedia.com/terms/e/ethereum.asp.

[9] Adam Hayes. Blockchain explained, Apr 2022. URL https://www.investopedia.com/terms/b/blockchain.asp.

[10] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, Mar 2009. URL https://bitcoin.org/bitcoin.pdf.

[11] Caitlin Ostroff i Santiago Pérez. El salvador becomes first country to approve bitcoin as legal tender, Jun 2021. URL https://www.wsj.com/articles/el-salvador-becomes-first-country-to-approve-bitcoin-as-legal-ten

[12] Randao. Randao/randao: Randao: A dao working as rng of ethereum. URL https://github.com/randao/randao.

Sigurnosne ranjivosti u pametnim ugovorima

Sažetak

Rad obrađuje nekolicinu odabranih ranjivosti u implementaciji pametnih ugovora na platformi Ethereum. Za svaku razrađuje uzroke i preventivne tehnike. U nastavku analizira problem generiranja slučajnih brojeva i osvrće se na loše prakse u tom procesu. Uz to prezentira rješenje za pronalaženje loših praksi u implementaciji ugovora. Navedeno rješenje se potom testira kroz nekolicinu primjera i razmatraju se daljnje nadogradnja. Na kraju se predlažu dobre prakse za dobivanje slučajnih brojeva.

Ključne riječi: Bitcoin, lanac blokova, Ethereum, pametni ugovor, ranjivosti, slučajni brojevi, statička analiza koda

Security vulnerabilities in smart contracts

Abstract

This paper addresses several selected vulnerabilities in the implementation of smart contracts on the Ethereum network. For each, it examines causes and presents some preventive techniques. In the following, it analyzes the problem of generating random numbers and examines bad practices during the implementation process. In addition, it presents the solution for finding those practices in contract implementation. That solution is then tested through several test examples and further upgrades are considered. Finally, good practices for obtaining random numbers are suggested.

Keywords: Bitcoin, blockchain, Ethereum, smart contract, vulnerability, random numbers, static code analysis