VIŠJA STROKOVNA ŠOLA ACADEMIA

MARIBOR

VARNOST PAMETNIH POGODB, NAPISANIH V PROGRAMSKEM JEZIKU SOLIDITY

Kandidatka: Petra Munda

Vrsta študija: študentka izrednega študija

Študijski program: Informatika

Mentor predavatelj: mag. Ervin Schaff

Mentorica v podjetju: Monika Mastnak, dipl. inž. inf. in tehnol. kom. (UN)

Lektorica: dr. Aleksandra Gačić Belej, univ. dipl. prof. zgo. in slov.

Maribor, 2022

IZJAVA O AVTORSTVU DIPLOMSKEGA DELA

Podpisana Petra Munda sem avtorica diplomskega dela z naslovom Varnost pametnih pogodb, napisanih v programskem jeziku Solidity, ki sem ga napisala pod mentorstvom mag. Ervina Schafa.

S svojim podpisom zagotavljam, da:

- je predloženo delo izključno rezultat mojega dela,
- sem poskrbela, da so dela in mnenja drugih avtorjev, ki jih uporabljam v predloženi nalogi, navedena oz. citirana skladno s pravili Višje strokovne šole Academia Maribor,
- se zavedam, da je plagiatorstvo – predstavljanje tujih del oz. misli, kot moje lastne kaznivo po Zakonu o avtorski in sorodnih pravicah (Uradni list RS, št. 16/07 – uradno prečiščeno besedilo, 68/08, 110/13, 56/15 in 63/16 – ZKUASP); prekršek pa podleže tudi ukrepom Višje strokovne šole Academia Maribor skladno z njenimi pravili,
- skladno z 32.a členom ZASP dovoljujem Višji strokovni šoli Academia Maribor objavo diplomskega dela na spletnem portalu šole.

Maribor, avgust 2022 Podpis študentke:

ZAHVALA

Zahvaljujem se mentorjema mag. Ervinu Schaffu in Moniki Mastnak, dipl. inž. inf. in tehnol. kom. (UN), za strokovno pomoč in ves vloženi čas pri izdelavi diplomskega dela ter mentorstvu v podjetju. Zahvaljujem se tudi vodstvu Višje strokovne šole Academia Maribor in vsem predavateljem za vsa posredovana znanja in odlično organizacijo študija.

Prav tako se zahvaljujem svojemu partnerju Marku in mami Nadi za vso dano podporo med študijem ter spodbudo pri pisanju diplomskega dela.

POVZETEK

Kodiranje pametnih pogodb v programskem jeziku Solidity predstavlja poseben izziv, saj posebnosti tehnologije veriženja blokov zahtevajo uporabo nekonvencionalnih programerskih vzorcev. Morebitni programski hrošči in varnostne pomanjkljivosti, zapisani v programski kodi, so vidni vsem, saj je izvorna koda pametnih pogodb javna. Z vidika varnosti pametnih pogodb je zato priporočljivo, da ima programer dobro poznavanje uveljavljenih vzorcev načrtovanja, ki omogočajo pisanje funkcionalne programske kode z zmanjšano možnostjo delanja napak.

V diplomskem delu obravnavamo razliko med različnimi vrstami algoritmov porazdelitvenega soglasja in navajamo osnovne gradnike verig blokov. Z navedbo konkretnih primerov uporabe verig blokov v realnih poslovnih situacijah smo želeli razbliniti predstavo o tem, da so pametne pogodbe uporabne samo na področju finančnih transakcij. V nadaljevanju se v okviru diplomskega dela ukvarjamo s posebnimi lastnostmi programskega jezika Solidity, opredelimo njegove prednosti in slabosti. Kot eno od večjih slabosti programskega jezika Solidity lahko vsekakor poudarimo njegovo mladost in s tem povezano relativno veliko število varnostnih pomanjkljivosti ter pogostih sprememb in nadgradenj programske kode. Iz tega razloga navajamo največje in najpogostejše ranljivosti pametnih pogodb, napisanih v Solidity. Za vsako opisano ranljivost poleg konkretnega primera podajamo še priporočila, kako se tem ranljivostim izogniti ali jih vsaj zmanjšati. Kjer je bilo to mogoče, smo s konkretnimi primeri, zapisanimi s programsko kodo, podali primer ranljive programske kode in primer, kjer smo to ranljivost odpravili.

Na osnovi podanih ranljivosti in priporočil smo pripravili in v diplomsko delo vključili tudi kontrolni seznam, ki bi programerjem v prihodnje lahko služil kot hitra referenca pri njihovem delu. Kontrolni seznam smo zasnovali tako, da smo pri vsaki ranljivosti navedli področje, na katero se le-ta nanaša, ter tudi kratko navedbo v zvezi s priporočenimi in odsvetovanimi praksami. Na tak način smo želeli programerjem še dodatno poenostaviti uporabo kontrolnega seznama.

Pridobljena spoznanja smo nato vključili v pisanje lastnega primera aplikacije, in sicer tržnice NFT-žetonov, ki uporabniku omogoča povezavo aplikacije z denarnico in prodajo žetonov. Aplikacija je sestavljena iz dveh pametnih pogodb in preprostega grafičnega uporabniškega vmesnika, napisanega v JavaScriptu.

Ključne besede: Solidity, pametne pogodbe, ranljivosti, verige blokov, Ethereum

SUMMARY

Security of smart contracts written in the Solidity programming language

Coding smart contracts in the Solidity programming language represents a special challenge, as the specifics of blockchain technology require the use of unconventional programming patterns. Any bugs and security flaws written in the code are visible to everyone, as the source code of smart contracts is public viewable. From the security point of view, it is therefore recommended that the Solidity programmer should have a good knowledge of established programming design patterns. That allows him or her to write functional programming code and reduces the possibility of making mistakes.

In our thesis, we discuss the difference between different types of distributed consensus algorithms and list the basic building blocks of blockchains. By listing concrete examples of blockchain use in real business situations, we wanted to showcase that smart contracts are not only used in the field of financial transactions.

Further we deal with the specific characteristics of the Solidity programming language and define its strengths and weaknesses. As one of the major weaknesses of Solidity, we can certainly point out its young age, as well as the related relatively large number of security flaws, frequent changes, and upgrades of the program Solidity programming language.

For this reason, we listed the biggest and most common vulnerabilities of smart contracts written in the Solidity programming language. For each described vulnerability, we quoted an example and provided recommendations on how to avoid or at least reduce these vulnerabilities. Where possible we provided also examples of vulnerable code and code where we tried to fix the quoted vulnerability.

Based on the listed vulnerabilities and given recommendations, we prepared a checklist, which could serve as a quick reference for programmers when coding smart contracts. For further simplification of the checklist usage for each vulnerability we provided a field, which it relates to. For that reason, we also included in the checklist aa brief statement regarding recommended and discouraged practices.

We then incorporated that knowledge into writing our own example application, an NFT token marketplace, allowing the users to connect their wallets and sell their tokens. The application consists of two smart contracts and a simple graphical user interface written in JavaScript.

Keywords: Solidity, smart contracts, vulnerabilities, blockchains, Ethereum

KAZALO VSEBINE

1 UVOD 12
1.1 Opis področja in opredelitev problema 12
1.2 Namen, cilji in osnovne trditve 13
1.3 Predpostavke in omejitve 15
1.4 Uporabljene raziskovalne metode 15
1.5 Motivacija 16
2 VERIGE BLOKOV 17
2.1 Algoritmi porazdeljenega soglasja 18
2.2 Dokazilo o delu 18
2.3 Dokazilo o deležu 18
2.4 Delegirano dokazilo o deležu 19
2.5 Osnovni gradniki 19
2.6 Primeri uporabe 19
3 PAMETNE POGODBE 22
3.1 Programski jezik Solidity 22
3.1.1 Niz znakov (angl. String) 23
3.1.2 Dogodki (angl. Events) 23
3.1.3 Zanke for loop 24
3.1.4 Preslikave (angl. Mapping) 24
3.1.5 Modifikatorji dostopa 25
3.1.6 Napake prekoračitve (angl. Overflow errors) 26
3.1.7 Optimizacija 26
3.1.8 Podatkovne strukture in abstraktni tipi podatkov 26
3.1.9 Byte in bytes 27
3.2 Decentralizirane programske aplikacije 27
4 VARNOST PAMETNIH POGODB 28
4.1 Ranljivosti funkcij in metod 28
4.1.1 Privzeta vidnost funkcij (angl. Function Default Visibility) 28
4.1.2 Nezaščiten klic funkcije za samouničenje (angl. Unprotected Selfdestruct Instruction) 29
4.1.3 Uporaba zastarelih funkcij (angl. Use of Deprecated Solidity Functions) 30
4.1.4 Nepravilno poimenovanje konstruktorja (angl. Incorrect Constructor Name) 30
4.1.5 Nepravilen vrstni red dedovanja (angl. Incorrect Inheritance Order) 32
4.1.6 Nezaščiten dvig Ethra (angl. Unprotected Ether Withdrawal) 32
4.2 Ranljivosti spremenljivk 33

4.2.1 Privzeta vidnost spremenljivk stanja (angl. State Variable Default Visibility) ... 33
4.2.2 Napaka prekoračitve in podkoračitve celih števil (angl. Integer Overflow and Underflow) ... 33
4.2.3 Neinicializiran kazalec v pomnilniku (angl. Uninitialized Storage Pointer) ... 35
4.2.4 Avtorizacija s tx.origin (angl. Authorization through tx.origin) ... 35
4.3 Ranljivosti v povezavi s klici ... 35
4.3.1 Nepreverjene vrnjene vrednosti klicev (angl. Unchecked Call Return Value) ... 35
4.3.2 Ponovni vstop (angl. Reentrancy) ... 36
4.3.3 Delegiran klic na nepreverjeno pogodbo (angl. Delegatecall to Untrusted Callee) ... 38
4.4 Ranljivosti v povezavi z gasom ... 39
4.4.1 Nezadostna količina gasa (angl. Insufficient Gas Griefing) ... 39
4.4.2 Fiksno kodiran znesek gasa funkcij (angl. Message call with hardcoded gas amount) ... 40
4.4.3 DoS-napad s pomočjo omejitve gasa bloka (angl. DoS With Block Gas Limit) ... 43
4.5 Druge ranljivosti ... 43
4.5.1 Določanje časa iz časovnih vrednosti blokov (angl. Block values as a proxy for time) ... 43
4.6 Kontrolni seznam ... 43
4.7 Varnostne ranljivosti in rešitve, objavljene s strani organizacije Ethereum ... 48
4.8 Posodobitve programskega jezika Solidity ... 48
4.9 Razprava s področja varnostnih ranljivosti Solidity ... 49
4.10 Pametne pogodbe in zaupanje v njihovo varnost ... 50
5 KODIRANJE PAMETNIH POGODB KOT DELO OD DOMA ... 52
6 PRIMER REŠITVE ... 55
6.1 Izvorna koda pogodbe NFT.sol ... 55
6.2 Izvorna koda pogodbe NFTTrznica.sol ... 56
6.2.1 Funkcija createMarketItem ... 57
6.2.2 Funkcija createMarketSale ... 59
6.2.3 Druge funkcije ... 60
6.3 Izziv ... 61
6.4 Testiranje delovanja pametnih pogodb s pomočjo Chai in Hardhat ... 61
7 SKLEP ... 65
8 VIRI IN LITERATURA ... 69
9 PRILOGE ... 72
9.1 Tabela različic Solidity v primerjavi z različicami programskega jezika Kotlin ... 72
9.2 Izvorna koda aplikacije ... 75
9.2.1 Pametna pogodba NFT.sol ... 75
9.2.2 Pametna pogodba NFTTrznica.sol ... 76
9.2.3 Skripta za deploy pametnih pogodb na verigi blokov deploy.js ... 81

9.2.4 Izgled grafičnega vmesnika 81

KAZALO SLIK

Slika 1: manipulacij nizov znakov 23
Slika 2: Dogodki v Solidity 23
Slika 3: Zanke for loop 24
Slika 4: Preslikave 25
Slika 5: Ranljivost privzete vidnosti funkcij 29
Slika 6: Ranljivost nazaščitenega klica funkcije za samouničenje 29
Slika 7: Odpravljena ranljivost nezaščitenega klica funkcije za samouničenje 30
Slika 8: Ranljivost nepravilnega poimenovanja konstruktorja 31
Slika 9: Odpravljena ranljivost nepravilnega poimenovanja konstruktorja 31
Slika 10: Večstopenjsko dedovanje 32
Slika 11: Odpravljena ranljivost nezaščitenega dviga Ethra 33
Slika 12: Ranljivost napake prekoračitve in podkoračitve 34
Slika 13: Odpravljena ranljivost napake prekoračitve in podkoračitve 34
Slika 14: Ranljivost nepreverjene vrnjene rednosti klicev 36
Slika 15: Odpravljena ranljivost nepreverjene vrnjene rednosti klicev 36
Slika 16: Shema poteka napada pri ranljivosti ponovnega vstopa 37
Slika 17: Popravljena ranljivost ponovnega vstopa 38
Slika 18: Ranljivost nezadostne količine gasa 40
Slika 19: Popravljena ranljivost nezadostne količine gasa 40
Slika 21: Metamask (vse možnosti izbire zneska gasa) 42
Slika 22: Izvorna koda NTF.sol 55
Slika 23: Izvorna koda NTF.sol - ustvarjanje novih žetonov 56
Slika 24: Izvorna koda NTFTrznica.sol 57
Slika 25: Izvorna koda NTFTrznica.sol - ustvarjanje NTF izdelka 59
Slika 26: Izvorna koda NTFTrznica.sol - ustvarjanje prodaje 60
Slika 27: Rezultat testa pravilnega delovanja pametne pogodbe NFTTrznica.sol – prikaz 1 62
Slika 28: Rezultat testa pravilnega delovanja pametne pogodbe NFTTrznica.sol - prikaz 2 63
Slika 29: Izvorna koda testa za NTFTrznica.sol 64
Slika 30: Prikaz gčavne strani grafičnega vmesnika aplikacije 81
Slika 31: Prikaz strani "Prodaja" grafičnega vmesnika 82
Slika 32: Prikaz strani "Moji dodani NFT-ji" 82
Slika 33: Prikaz NFT-jev, ki jih je dodala denarnica Account 2 83

Slika 34: Prikaz NFT-jev, ki jih je dodala denarnica Account 3 83

KAZALO TABEL

Tabela 1: Kontrolni seznam za preverjanje ranljivosti pametnih pogodb 45

RAZLAGA STROKOVNIH IZRAZOV IN KRATIC

Application Binary Interface (ABI) – aplikacijski binarni vmesnik predstavlja vmesnik med dvema binarnima programskima moduloma, od katerih je pogosto en knjižnica ali operacijski sistem in drugi program, ki ga izvaja uporabnik. ABI opredeli, kako se v strojni kodi dostopa do podatkovnih struktur in postopkov.

C3 pravilo linearizacije – algoritem, ki se uporablja za določanja vrstnega reda večstopenjskega dedovanja.

Decentralizirane aplikacije (Dapp) – aplikacije, ki delujejo v porazdeljenem omrežju (P2P, tj. enak z enakimi ali v verigi blokov).

Decentralizirane finančne aplikacije (DeFi) – decentralizirane aplikacije s finančnega področja.

Delegirano dokazilo o deležu (angl. Delegated Proof of Stake, DpoS) – vrsta protokola za doseganje soglasja v verigi blokov. Ta uporabnikom omogoča, da na osnovi svojega deleža kripto žetonov glasujejo za določenega delegata. Izvoljeni delegati lahko nato dosegajo kritične odločitve v zvezi z omrežjem. DpoS spodbuja uporabnike, da z založenim vložkom (v obliki kriupto žetonov) zavarujejo omrežje in delujejo v njegovo korist.

Dokazilo o deležu (angl. Proof of Stake) – vrsta protokola za doseganje soglasja, ki temelji na višini deleža kripto žetonov (ali valute) posameznika.

Ethereum – decentralizirano omrežje verige blokov.

Ether (ETH) – kripto žetoni (ali valuta), ki se uporabljajo znotraj omrežja Ethereum.

Ethereum Virtual Machine (EVM) – omrežje računalnikov znotraj Ethereum omrežja, ki deluje kot decentraliziran računalnik in skrbi za izvajanje programske kode, med drugim tudi pametnih pogodb.

Množično financiranje (angl. Crowdfunding) – način financiranja podjetniških projektov z zbiranjem denarja za zagonska sredstva.

Non-fungible token (NFT) – unikatno digitalno sredstvo, katerega HASH-zapis in tudi lastništvo tega sredstva v kripto denarnici sta shranjena v verigi blokov.

Pametna pogodba – programska logika decentralizirane aplikacije, ki se izvaja na Ethereum Virtual Machine na verigi blokov in je v primeru Ethereum omrežja napisana v programskem jeziku Solidity.

Proof of Work (PoW) – vrsta protokola za doseganje soglasja, ki temelji na računalniški moči rudarjev.

Web3 (tudi Web 3.0) – nova generacija WWW-protokola, na katerem temelji veriga blokov, pri katerem je v ospredju medsebojna komunikacija različnih prenosnih naprav z dostopom do interneta.

1 UVOD

1.1 Opis področja in opredelitev problema

Platforma verig blokov Ethereum po merilu tržne kapitalizacije zaseda drugo mesto takoj za Bitcoinom. V primerjavi z Bitcoinom omogoča omrežje Ethereum s pomočjo Ethereum Virtual Machine (EVM) in knjižnico Web3.js razvoj decentraliziranih programskih aplikacij. EVM lahko opišemo kot decentraliziran računalnik, čigar programska koda se izvaja na številnih vozliščih oz. računalnikih. Med decentraliziranimi aplikacijami so v preteklem letu največji razvoj doživele decentralizirane finančne aplikacije (DeFi). Za decentralizirane aplikacije je značilno, da jih ne upravlja nobena centralna avtoriteta, ki bi lahko cenzurirala ali spreminjala pogoje uporabe ali samo programsko kodo takšne aplikacije. Programske kode decentralizirane aplikacije po tem, ko smo jo naložili na verigo blokov, namreč ne moremo več spreminjati. Spremenimo jo lahko le posredno v smislu, da staro pametno pogodbo uporabimo kot proxy za novo pogodbo, vendar smo morali to funkcionalnost predvideti v prvotni pogodbi. Ključni del decentraliziranih aplikacij so pametne pogodbe, tj. programska logika, ki so za Ethereumovo verigo blokov napisane v programskem jeziku Solidity.

Pametne pogodbe prek decentraliziranih finančnih aplikacij v Ethereumovem ekosistemu ustvarijo mesečno na desetine milijard evrov prometa. Ker je programska koda pametnih pogodb naložena na verigo blokov, je tako kot veriga javna in vidna vsakomur. Prav tako so vsakomur vidne vse transakcije in stanje sredstev v sleherni denarnici oz. generiranem javnem ključu. V primeru napake ali hekerskega vdora sredstev ni moč povrniti, saj so transakcije nepovratne. Vsakogar, ki se ukvarja z varnostjo informacijskega sistema v banki ali drugi finančni instituciji, bi ob vsem naštetem prav gotovo zabolela glava. Vendar je opisani scenarij realnost, s katero se vsakodnevno srečujejo razvijalci pametnih pogodb. Odprtokodni sistem, nespremenljivost programske kode in transparentnost transakcij so lastnosti, ki so tej novi industriji zagotovo dale veliko zagona, po drugi strani so ji prinesle tudi večji dejavnik tveganja za morebitne hekerske vdore. Zaradi navedenega morajo biti razvijalci pri kodiranju pametnih pogodb posebej pozorni na morebitne ranljivosti programske kode.

Hekerski napadi na pametne pogodbe so bili v preteklosti ne samo številčni, temveč žal tudi uspešni. Najodmevnejša napada sta bila tako imenovana DAO-napad in Parity Wallet vdor, s katerima so napadalci uspeli odtujiti na desetine milijonov evrov. Varnostne dejavnike tveganja pametnih pogodb opisujejo številni avtorji. Tako recimo Sayeed, Marco-Gisbert in Caira (2020) ločijo med štirimi klasifikacijami in sedmimi tehnikami napadov na pametne pogodbe. V prispevku *Smart Contract: Attacks and Protections* omenjeni avtorji vidijo rešitev za čim večjo

stopnjo zaščite v uporabi posebnih orodij za statično in dinamično analizo varnosti pametnih pogodb, kot so recimo Securify, Mythril, Echida in druge. Na drugi strani skušata recimo Wörther in Zdun (2018) v svoji raziskavi opredeliti varnostne vzorce pri kodiranju pametnih pogodb. V svojem znanstvenem delu skušata odgovoriti na vprašanje, kateri vzorci načrtovanja so najpogosteje uporabljeni v Solidity pametnih pogodbah. Prav tako obstajajo tudi podjetja, ki se ukvarjajo z varnostjo pametnih pogodb. Tako je recimo podjetje OpenZeppelin razvilo ogrodje za kodiranje nadgradljivih in varnih pametnih pogodb. Podjetje ConsenSys Dilligence pa redno objavila varnostne prakse.

Organizacija Ethereum varnostne ranljivosti programskega jezika Solidity na svojih spletnih straneh spremlja oz. objavlja le do neke mere. Trenutno na njihovi spletni strani ni objavljenega seznama vseh obstoječih ranljivosti. Organizacija Ethereum sicer navaja nekaj primerov dobrih praks, vendar bralca kljub temu za podrobnejši pregled ranljivosti in priporočil napotuje na spletne strani drugih avtorjev. Seznam znanih programskih hroščev Solidity je prav na spletni strani Organizacije Ethereum objavljen v obliki JSON-a, kar za bralca zagotovi ni najprijaznejša oblika pregledovanja vsebin.

1.2 Namen, cilji in osnovne trditve

Cilj tega diplomskega dela je poiskati in opredeliti konkretne varnostne mehanizme ter varnostna priporočila za pisanje pametnih pogodb v programskem jeziku Solidity. Predvsem nas zanimajo morebitne rešitve, ki so bile ugotovljene po že izvedenih hekerskih napadih. V diplomskem delu primerjamo smernice in rešitve za konkretne težave oz. obstoječa varnostna tveganja, navedena v literaturi. Cilj tega diplomskega dela je torej preučiti, na kakšen način se je pri pisanju pametnih pogodb v programskem jeziku Solidity moč izogniti omenjenim varnostnim tveganjem in s tem zmanjšati možnost izpostavljenosti hekerskim napadom.

Cilj tega diplomskega dela je na enem mestu združiti vsa pomembna varnostna pravila, ki bi jih moral poznati vsak Solidity razvijalec. S pomočjo lastnih primerov smo v empiričnem delu podali priporočila glede varne programske kode v programskem jeziku Solidity. Takšen celovit pregled bi razvijalcem lahko služil kot neke vrste referenčna lista, zato smo za te namene pripravili tudi kontrolni obrazec, s katerim bi lahko kot razvijalci hitro preverili, ali smo pri razvoju aplikacije oz. pametne pogodbe upoštevali potrebne varnostne smernice.

V empiričnem delu smo prav tako razvili manjšo decentralizirano aplikacijo za izbrani scenarij, napisano v programskem jeziku Solidity. Cilj aplikacije je predvsem pokazati, kako je treba pri pisanju pametnih pogodb upoštevati pomembne varnostne smernice.

V okviru tega diplomskega dela smo si odgovorili na naslednja raziskovalna vprašanja in hipoteze:

- **Raziskovalno vprašanje 1:** Ali obstajajo varnostna tveganja pametnih pogodb v Solidity?
  - **Raziskovalno podvprašanje 1.1:** Kakšna so najpogostejša varnostna tveganja pametnih pogodb?
  - **Raziskovalno podvprašanje 1.2:** Kako lahko varnostna tveganja pametnih pogodb zmanjšamo?
- **Raziskovalno vprašanje 2:** So pametne pogodbe vredne zaupanja?
  - **Raziskovalno podvprašanje 2.1:** Kakšne preventivne ukrepe bi bilo dobro sprejeti pri kodiranju pametnih pogodb?
- **Raziskovalno vprašanje 3:** Ali je kodiranje pametnih pogodb iskano kot delo od doma?
- **Hipoteza H1:** Pametne pogodbe so načeloma varne, vendar kljub temu vsebujejo tveganja, ki jih ne bi smeli prezreti.
- **Hipoteza H2:** Ethereum Fundation pogosto posodablja programski jezik Solidity, s čimer pripomore k zagotavljanju večje varnosti pametnih pogodb.
- **Hipoteza H3:** Predvidevamo, da bomo lahko različna varnostna priporočila, kako pisati varne pametne pogodbe, združili v en dokument, ter najpomembnejša implementirali v lastni aplikaciji za izbrani scenarij.
- **Hipoteza H4:** Prehod iz programskega jezika JavaScript v programski jezik Solidity je dokaj enostaven**.**
- **Hipoteza H5:** Za kodiranje pametnih pogodb je na voljo veliko delovnih mest za delo od doma.

Namen tega diplomskega dela tako ni zajeti vseh obstoječih varnostnih tveganj in hroščev, saj bi zaradi njihovega števila to preseglo zastavljene okvirje in smernice glede priprave diplomskega dela, temveč je namen zajeti tista varnostna tveganja, ki so najpogostejša in jih združiti v pregledno obliko, ki bo lahko v pomoč razvijalcem pri kodiranju pametnih pogodb v Solidity.

Z *raziskovalnim vprašanjem 1* se ukvarjamo v četrtem poglavju. V tem poglavju navedemo in opišemo tudi najpogostejša varnostna tveganja pametnih pogodb in programskega jezika Solidity ter tako odgovorimo na prvo raziskovalno podvprašanje (*1.1*). Z drugim raziskovalnim podvprašanjem (*1.2*) se ukvarjamo pri predstavitvi varnostnih tveganj. Le-ta razdelimo v pet kategorij in k vsakemu opisanemu varnostnemu tveganju podamo tudi priporočila, kako to tveganje zmanjšati.

Na *raziskovalno vprašanje 2* in njegovo podvprašanje (*2.1*) odgovorimo v poglavju 4.10. Podrobneje se s posameznimi varnostmi ukrepi ukvarjamo predvsem v podpoglavjih od 4.1 do 4.7 in tako pokrijemo vse segmente *raziskovalnega vprašanja 2*.

Z *raziskovalnim vprašanjem 3* se ukvarjamo v petem poglavju, kjer opravimo krajši pregled trga dela za področje kodiranja pametnih pogodb.

Na hipotezo H1 smo odgovorili in jo potrdili v poglavju 4.9, medtem ko smo se s hipotezo H2 ukvarjali v poglavju 4.8. Na hipotezi H3 in H4 smo podali odgovor v sedmem poglavju, medtem ko smo hipotezo H5 v celoti potrdili v petem poglavju.

1.3 Predpostavke in omejitve

Pri pisanju diplomskega dela smo se omejili na programski jezik Solidity in platformo verig blokov Ethereum. Aplikacijo smo testirali na lastnem omrežju verige blokov, tako imenovani lokalni verigi blokov. Dodatna omejitev pri pisanju tega diplomskega dela je bila tudi uporaba razvojnih orodij, ki niso plačljiva. V primeru aplikacije smo se omejili na področje pametnih pogodb za trgovanje z NFT-žetoni.

1.4 Uporabljene raziskovalne metode

V diplomskem delu smo uporabili več raziskovalnih metod. V teoretičnem delu smo uporabili **metodo analize**, s katero smo analizirali priporočila in predloge različnih avtorjev, ter **komparativno metodo**, s pomočjo katere smo poiskali skupne točke v obstoječi literaturi in smernice ter jih med sabo primerjali. Pri tem smo se osredotočili tudi na kompromise in na razlike med posameznimi metodami zagotavljanja varnosti pametnih pogodb. Za ti dve metodi smo se odločili, ker je področje raziskovanja še dokaj mlado in se nam zdi smiselno pripraviti primerjalni pregled obstoječih rešitev. V empiričnem delu diplomskega dela, v katerem smo analizirali oz. pregledali trg delovne sile in možnosti za delo od doma za razvijalce pametnih pogodb, smo uporabili **kvantitativen raziskovalni pristop** in rezultate analize predstavili z **deskriptivno metodo**. V empiričnem delu smo na realnem izzivu naredili prototip aplikacije in s pomočjo **metode abstrakcije** predstavili bistvene lastnosti rešitve.

Področje raziskovanja lahko opredelimo kot:

- področje varnosti in ranljivosti pametnih pogodb za morebitne hekerske napade ter
- področje stanja na trgu dela za razvijalce pametnih pogodb v smislu možnosti in primernosti za delo od doma.

1.5 Motivacija

Tehnologija veriženja blokov in razvoj aplikacij na tem področju sta v zelo velikem porastu. Po eni strani je Solidity razvijalcev glede na potrebe trga še relativno malo, zaradi česar so možnosti za vključitev na trg delovne sile za junior razvijalce v primerjavi z vključitvijo v razvoj spletnih aplikacij večje. Po drugi strani prehod iz programskega jezika JavaScript v programski jezik Solidity, ob poznavanju določenih lastnosti in posebnosti sintakse Solidity, ni zahteven. Solidity je bil namreč razvit tudi z namenom, da bi ga uporabljalo čim večje število programerjev. Razvoj programske kode pametnih pogodb na osnovi tehnologije veriženja blokov prinaša povsem nove možnosti in pristope k reševanju realnih življenjskih problemov. Za uspešnost tovrstnih rešitev je ključno poznati načine, kako zmanjšati varnostna tveganja napisanih pametnih pogodb.

2 VERIGE BLOKOV

Verige blokov so digitalna tehnologija, ki sloni na kombinaciji kriptografije, računalniških mrež ter mehanizmov za preverjanje, potrjevanje in beleženje transakcij med različnimi uporabniki, ki si med sabo pošiljajo in prejemajo določeno kriptografsko valuto oz. kovanec. Najbolj znane kriptografske valute so Bitcoin, Ethereum in Litecoin. Verige blokov lahko opišemo tudi kot decentralizirano zbirko podatkov, ki je kriptografsko varovana in tako onemogoča spreminjanje že potrjenih transakcij. Veriga blokov rešuje problem tako imenovane dvojne porabe (angl. *Dubble spending*). Na ta način skuša veriga blokov vzpostaviti zaupanje med uporabniki, brez vključitve tretje osebe, tj. banke ali druge finančne ustanove (Wörther in Zdun, 2018). Začetki verig blokov, ki so kriptografsko varovane z zgoščevalno funkcijo, segajo v leto 1991, ko sta Haber in Stornetta (1991) zaradi pojava digitalizacije opisala težavo glede verodostojnosti, kdaj je bil določen dokument ustvarjen ali spremenjen. Avtorja sta rešitev videla v tem, da je treba s časovnimi žigi opremiti dokumente in ne nosilcev, na katerih se ti dokumenti hranijo. Dokumente je nato treba še digitalno podpisati. Da bi preprečila možnosti manipulacije časovnih žigov, sta Harber in Stonetta predlagala, da bi vsak novi digitalno podpisan dokument vseboval tudi časovni žig predhodnega dokumenta. Svoj koncept sta avtorja Haber in Stornetta (1991) dve leti kasneje nadgradila z uporabo Merklovega drevesa za združevanje večjega števila podatkov v en sam blok in tako hitrejšega delovanja verige.

Leta 2008 je razvijalec ali skupina razvijalcev pod psevdonimom Satoshi Nakamoto razvila prvo platformo, zgrajeno na verigi blokov, imenovano Bitcoin. V svojem prispevku je Nakamoto (2008) predstavil nov elektronski denarni sistem, ki temelji na omrežju Peer to peer oz. na omrežju računalnikov, ki deluje brez centralnega posrednika. V prispevku je Nakamoto zapisal, da je temeljni problem za pravilno delovanje običajnih valut v zaupanju njenih uporabnikov. Centralne banke imajo v rokah škarje in platno, da ta sistem manipulirajo in za obdelavo plačil zaračunavajo poljubne provizije in za njihovo izvedbo postavljajo lastne pogoje. Zato Nakamotova rešitev namesto na zaupanju do tretjih finančnih entitet temelji na kriptografskem dokazu in neposrednih transakcijah med dvema akterjema. Te transakcije, poslane s strani uporabnikov, so vključene v verige blokov. Vozlišča nato te bloke skušajo dodati verigi blokov. Vozlišče, ki mu uspe kot prvemu potrditi nov dodani blok, je za svoje delo nagrajeno v obliki kriptovalute. Ta proces se imenuje tudi rudarjenje, medtem ko sodelujoča vozlišča imenujemo rudarji.

2.1 Algoritmi porazdeljenega soglasja

Verige blokov delujejo kot decentralizirani, samoregulacijski sistemi brez kakršne koli avtoritete Frankenfield in Mansa (2021). Da bi bila transakcija med dvema akterjema brez posredovanja tretje entitete uspešna, je torej potreben konsenz. Ta konsenz je dosežen s pomočjo programske kode oz. algoritma porazdeljenega soglasja, ki predstavlja zaporedje pravil za določanje legitimacije transakcije. Legitimacijo transakcije v novem bloku verificirajo in avtenticirajo rudarji ali validatorji. Poznamo vsaj štirinajst različnih algoritmov porazdeljenega soglasja, vendar v okviru tega diplomskega dela ne bomo obravnavali vseh. Najbolj znana sta *dokazilo o delu (angl. Proof of Work)* ter *dokazilo o deležu (angl. Proof of Stake)*.

2.2 Dokazilo o delu

Algoritem porazdeljenega soglasja *dokazilo o delu (angl. Proof of Work, ali PoW)* je prvi implementiral Nakomoto pri kriptovaluti Bitcoin. PoW deluje na načelu, da rudarjem omogoča potrjevanje novih blokov transakcij v verigo blokov, če rešijo kriptografsko uganko. Pri tem gre za iskanje naključnega števila (angl. *Nonce*), ki bo skupaj z vsemi v blok vključenimi podatki s pomočjo zgostitvene hash funkcije generiralo hash znakovni niz, ki ima na začetku niza določeno število ničel. Število potrebnih ničel na začetku se določi na osnovi težavnosti omrežja, tj. večja, kot je računska moč rudarjev, večja je težavnost in več ničel je potrebnih. PoW-algoritmi običajno uporabljajo SHA-256, SHA-3, Scrypt, Scrypt-n ali Scrypt-jane funkcije. Rudarji pri reševanju kriptografske uganke uporabljajo strojno opremo (CPU, grafične kartice, ASIC-čipe), zaradi česar je tovrstno potrjevanje blokov energetsko potratno (Sharma in Lal, 2020).

2.3 Dokazilo o deležu

Algoritem porazdeljenega soglasja *dokazila o deležu (angl. Proof of Stake, PoS)* predstavlja energetsko učinkovitejšo alternativo PoW-algoritmom in je bil prvič predstavljen leta 2012 s strani Sunny Kinga ter Scotta Nadala (Sharma in Lal, 2020). Za potrjevanje novih blokov transakcij v PoS vsak validator (analogija rudarjem v PoW) zastavi določeno število svojih kovancev. Ti zastavljeni kovanci so v denarnici validatorja ali na za to posebej določenem naslovu, zaklenjeni za neki omejen čas. Validator je izbran po deterministčnem načelu glede na količino zastavljenih kovancev. Pri PoS ni tako imenovane nagrade za novo generirani blok, temveč so validatorji plačani iz naslova plačanih provizij za validiranje transakcij.

2.4 Delegirano dokazilo o deležu

Pri *delegiranem dokazilu o deležu (angl. Delegated Proof of Stake, DpoS)* lastniki kovancev, ki bi želeli sodelovati v procesu validiranja, blokov le-teh ne validirajo sami, temveč za to izberejo delegata, ki to delo opravi namesto njih. Delegatov v omrežju je sicer omejeno število, a ker ima vsaka denarnica s kovanci pravico glasovati o delegatih, mehanizem ostaja decentraliziran. Prednost tega algoritma v primerjavi z dokazilom o deležu je v tem, da zaradi manjšega števila delegiranih validatorjev hitrost potrjevanja transakcij in skalabilnost ostajata visoki (Sharma in Lal, 2020).

2.5 Osnovni gradniki

Drugače kot pri klasičnih zbirkah podatkov, ki so centralizirano hranjene s strani ene entitete, ki to zbirko upravlja (recimo banke, ki hranijo na centralnem strežniku vse podatke o transakcijah svojih uporabnikov), se podatki v verigi blokov hranijo decentralizirano v obliki nespremenljive evidence zapisov. To v praksi pomeni, da so podatki o transakcijah zapisani v posameznem bloku v verigi blokov, to isto kopijo verige blokov posedujejo vsa vozlišča v omrežju, ki jih je več tisoč. Če bi posamezno vozlišče želelo manipulirati podatke v verigi blokov, bi o tem moralo prepričati več kot polovico vseh vozlišč, kar je pri veliki razširjenosti in decentralizaciji omrežij verig blokov skoraj nemogoče. Osnova za zaupanje v delovanje omrežja tako ni več neki zakonski akt, temveč tehnologija (Bär, 2017).

Glavne gradnike tehnologije verige blokov delimo na naslednje elemente, in sicer:

- *šifriranje in kriptografska orodja*: zagotavljajo identifikacijo, avtentikacijo in neoporečnost kopij verige blokov ter izvedenih transakcij,
- *mehanizem soglasja*: algoritem, ki prevzame vlogo odločevalca v omrežju glede tega, ali bo določen blok dodan v verigo blokov ali ne,
- *časovni žigi in zgoščevalne funkcije*: zagotavljajo, da vsak naslednji blok v verigi vsebuje niz znakov iz zgoščevalne funkcije iz podatkov prejšnjega bloka,
- *nespremenljiva evidenca zapisov*: veriga blokov, ki hrani podatke o zgodovini vseh transakcij v omrežju,
- *bazen transakcij (angl. Mempool):* skupek vseh veljavnih transakcij, ki čakajo v pomnilniku posameznih vozlišč, da jih rudarji ali validatorji vključijo v naslednji blok in s tem potrdijo.

2.6 Primeri uporabe

Od leta 2017, ko je začela delovati platforma Ethereum, je tehnologija verige blokov močno začela pridobivati pomembnost in število primerov uporabe. Če so bile pred tem obdobjem

kripto valute razumljene zgolj kot sredstvo hrambe in prenosa denarja brez centralne avtoritete, se je to po letu 2017 zelo spremenilo. Ethereum je javna in decentralizirana platforma, ki temelji na tehnologiji verige blokov in omogoča funkcionalnost pametnih pogodb. Pametne pogodbe se na platformi Ethereum izvajajo s pomočjo Ethereum navideznega stroja (angl. *Ethereum Virtual Machine* oz. *EVM*). EVM izvaja tako računske procese kot tudi zajema *stanje (angl. State)* pametnih pogodb. Uporablja niz navodil za strojni jezik, tako imenovanih opcode. Pametna pogodba se tako pretvori v niz opcode, ki računalniku daje navodila, kaj naj stori. EVM lahko razumemo kot globalni decentralizirani računalnik, na katerem se izvajajo vse pametne pogodbe. Kljub temu da se EVM vede kot ogromen računalnik, gre v bistvu za omrežje manjših računalnikov – vozlišč, ki med seboj stalno komunicirajo (Wörther in Zdun, 2018).

S pomočjo pametnih pogodb, zgrajenih na tehnologiji verige blokov, lahko avtomatiziramo in poenostavimo finančne storitve, kot so storitve menjalnic, izvedbo enostavnih in kompleksnih nakazil finančnih sredstev, množičnega financiranja *(angl. Crowdfunding)*, prav tako lahko avtomatiziramo odobritve finančnih posojil in vračila denarja. Na finančnem področju so v letu 2021 ogromen razvoj doživele decentralizirane finančne aplikacije, tj. decentralizirane spletne menjalnice in z njimi povezane storitve. Tehnologijo verige blokov (tukaj imamo v mislih predvsem verige blokov druge in tretje generacije, torej Ethereum, Solana, Avalanche, Terra, Cardano in druge) je možno uporabiti tudi na področju NFT-jev *(angl. Non-fungible tokens)*, tj. unikatnih digitalnih sredstev. Unikatna digitalna sredstva lahko uporabljamo za vse vrste digitalnih predmetov, za katere želimo uveljavljati lastninsko pravico in potrditev avtentičnosti. Tako lahko NFT-je uporabimo na področju digitalnih umetniških izdelkov (grafike, bitne slike, digitalne fotografije), digitaliziranih zapisov (glasba, govor, zapisana beseda) ter tudi predmetov, ki se uporabljajo v računalniških igrah (avatar, izkušnje, igričarski predmeti), in zbirateljskih predmetov (zbirateljske kartice, 3D-modeli, stripi).

Z razvojem omrežja Ethereum in pojavom njegovih konkurentov (Solana, Avalanche, Terra, Cardano …) se med razvijalci in snovalci omrežij verig blokov pojavljajo številne zamisli glede omenjene tehnologije.

V Dubaju so recimo mestne oblasti s pametnimi pogodbami povezale ulične svetilke. Ko senzorji v ulični svetilki zaznajo, da ta ne deluje, se zahtevek za popravilo pošlje v pogodbi določenemu serviserju. Ko senzorji svetilke zaznajo, da svetilka ponovno brezhibno deluje, se v pogodbi določen znesek za popravilo samodejno pošlje izvajalcu popravila (Khan, 2018).

Velik napredek pri integraciji tehnologije verige blokov pri digitalizaciji javne uprave je naredila tudi Estonija. Tako ima vsak Estonec danes nacionalno digitalno identiteto, varovano s tehnologijo verige blokov. S to digitalno identiteto se uporabniki lahko varno prijavljajo v vse vrste državnih servisov, od zdravstvenih storitev, bančnih storitev, se digitalno podpisujejo in sodelujejo v e-volitvah (Khan, 2018).

Primer uporabe pametnih pogodb prihaja tudi iz energetike, kjer je slovensko podjetje SunContract, ki na trgu nastopa kot distributer električne energije in ponudnik za izgradnjo sončnih elektrarn, svojim strankam (proizvajalcem in odjemalce) ponudilo nakup in prodajo električne energije na spletni borzi. Enote energije se trguje z žetoni SUN, ki so ERC-20-kovanci, izdani na platformi Ethereum. S tem poslovnim modelom je podjetje avtomatiziralo velik del poslovanja, ga pocenilo in s tem znižalo ceno električne energije za svoje stranke (Sonce, 2022).

3 PAMETNE POGODBE

Pametne pogodbe so računalniški programi, ki so namenjeni poenostavitvi, potrjevanju in izvedbi legalnih pogodb med udeleženimi strankami. Koda pametnih pogodb se izvaja kot del transakcije v verigi blokov, v katerih potekata prenos in prejem digitalne valute. Izvajanje pametne pogodbe se torej sproži s transakcijo ali ukazom druge pametne pogodbe. Ob tem sporočilu oz. transakciji se lahko izvede poljubno število ukazov, ki so lahko tudi zelo kompleksni. S pomočjo polja data, ki je del Ethereumovih transakcij, se pošiljajo vhodni parametri funkcijam pametnih pogodb. Polje data je rezervirano tudi za sprejem vrednosti, ki jih funkcije v pametnih pogodbah lahko vrnejo. Pametne pogodbe so napisane v domensko svojevrstnem jeziku. V primeru platforme Ethereum je to programski jezik Solidity.

Pametne pogodbe so običajno povezane z grafičnim vmesnikom (GUI), prek katerega uporabniki interagirajo s pametno pogodbo. S tem, ko pametno pogodbo naložimo na verigo blokov, ta postane avtonomna entiteta, ki avtomatizirano izvaja določene operacije, ko so izpolnjeni pogoji za to. Ker so pametne pogodbe naložene na verigo blokov, ki je decentralizirana, njihova koda pa je bila predhodno potrjena v določenem bloku, programske kode pametnih pogodb kasneje ni možno več spreminjati. Iz tega razloga pametnih pogodb, ker so del decentralizirane verige blokov, tudi ni možno cenzurirati ali prekiniti njihovega izvajanja s strani tretje osebe (Wörther & Zdun, 2018).

3.1 Programski jezik Solidity

Programski jezik Solidity je visokonivojski jezik, usmerjen v razvoj pametnih pogodb. Leta 2014 ga je razvil dr. Gavin Wood, soustanovitelj platforme Ethereum in ustanovitelj platforme Polkadot. Pri razvoju Solidity se je Wood delno orientiral po ECMA Scriptu, zaradi česar številni razvijalci trdijo, da je Solidity podoben Javascriptu, kljub temu, da ima največ konceptov iz C++ (Parizi, Singh in Dehghantanha, 2018). V programskem jeziku Solidity je treba tip spremenljivke določiti, medtem ko nam pri tem določanju tipa spremenljivke v Javascryptu ni treba biti tako natančen (zadoščata tipa *let* in const). Prednost statično tipiziranih programskih jezikov, kot je Solidity, je v primerjavi z dinamično tipiziranimi v tem, da se že v prevajalniku izvedejo številna preverjanja, zaradi česar lahko veliko trivialnih napak odkrijemo že zelo zgodaj. Nekatere posebnosti programskega jezika Solidity so opisane v naslednjih podpoglavjih tega poglavja.

3.1.1 Niz znakov (angl. String)

Solidity izvorno ne podpira manipulacij nizov znakov. V času pisanja tega diplomskega dela (različica Solidity 0.8.10) je celo za osnovne operacije, kot je priključitev enega niza znakov k drugemu, treba nize znakov spremeniti v polja bytov. Pretvarjanje v byte je sicer možno tudi s klicem funkcije *abi.encodePacked* iz Contract Application Binary Interface ABI (Ethereum, 2021). Primer funkcije, ki vrne dolžino niza znakov kot število:

```
contract MojaPogodba {
    function dolzina(string calldata str) external pure returns(uint) {
    return bytes(str).lenght;
    }
}
```

Slika 1: Manipulacij nizov znakov

Vir: Lasten

3.1.2 Dogodki (angl. Events)

Dogodki so v Solidity enosmerna vrsta komunikacije, in sicer od verige blokov do drugih aplikacij zunaj verige blokov (web frontend, mobilne aplikacije), ki lahko te dogodke poslušajo s pomočjo Web3.js knjižnice. Pametna pogodba ne more poslušati dogodkov, ne svojih in ne dogodkov v drugih pametnih pogodbah. Rezervirana beseda *indexed* omogoča filtriranje med več dogodki, vendar lahko v vsakem dogodku *indexed* uporabimo največ trikrat.

Primer dogodka – nakup kripto valute na decentralizirani borzi

```
contract MojaPogodba {
    event NovNakup (
        uint indexed datum,
        address indexed prodajalec,
        address indexed kupec,
        uint znesek
    );

    function posel(address kupec, uint znesek) external {
        emit NovNakup(now, msg.sender, kupec, znesek);
    }
}
```

Slika 2: Dogodki v Solidity

Vir: Lasten

3.1.3 Zanke for loop

Pri razvoju so se avtorji programskega jezika Solidity zgledovali po Javascriptu. Toda, ker se število nič, ki ga uporabljamo v for loop-ih, ko iterator postavimo na ničlo, nanaša na vrsto spremenljivke *byte* in ne na vrsto *integer*, se bo v Solidity v primeru, da je polje daljše od 255 elementov, iterator postavil nazaj na nič. Zato je treba *i* v Solidity deklarirati kot *uint* in ne kot *var*. Primer zanke for loop v Solidity:

```
for (uint i = 0; i < a.lenght; i++);
```

Slika 3: Zanke for loop

Vir: Lasten

3.1.4 Preslikave (angl. Mapping)

Če v Solidity pokličemo preslikan element, ki ima vrednost *null* ali ta element ne obstaja, ne dobimo vrnjene napake, temveč število 0. Tako ne moremo vedeti, ali gre za vrednost 0, ki je bila dodeljena elementu, ali gre za privzeto vrednost, ki je vrnjena, ker element v preslikavi ne obstaja. Solidity prav tako ne omogoča neposredne iteracije prek preslikanih elementov (kot je to možno recimo v Javi), prav tako tudi ne moremo s pomočjo zanke for loop priti do velikosti preslikave, kot to lahko storimo pri seznamu (angl. *Array*). To pomanjkanje funkcionalnosti lahko nadomestimo z implementacijo dodatnih preslikav in funkcij:

```
contract MojaPogodba {
      mapping(address => uint) public stanje;
      mapping(address => bool) public vstavljen; // spremlja, če je
      vstavljen ključ address iz 1. preslikave
      address[] public kljuci; // seznam ključev (address)

      // nastavimo stanje v preslikavi za elemente stanje

      function set(address _kljuc, uint _vrednost) external {
        stanje[_kljuc] = _vrednost // posodobimo preslikavo stanja

      // sledimo, ali je _kljuc bil vstavljen ali ne in če je bil ga
      vstavimo v seznam kljuci
      ff (!vstavljen[_kljuc]) {
        vstavljen[_kljuc] = true;
        kljuci.push(_kljuc);
        }
      }

      function getVelikost() external view returns (uint) {
        return kljuci.lenght;
      }

      function prviElement() external view returns (uint) {
        return stanja[kljuci[0]];
      }

      function zadnjiElement() external view returns (uint) {
        return stanja[kljuci[kljuci.lenght - 1]];
      }

      // pridobimo vrednost element i, ki je bil dodan v stanje

      function get(uint _i) external view returns (uint) {
        return stanje[kljuci[_i]];
}
```

Slika 4: Preslikave

Vir: Lasten

S klicem funkcije getVelikost in uporabo zanke for loop s funkcijo get lahko pri zgornjem primeru potem pridemo do velikosti preslikave.

3.1.5 Modifikatorji dostopa

Podobno kot v objektno orientiranih programskih jezikih poznamo tudi v Solidity modifikatorje dostopa, in sicer *public*, *external*, *internal* in *private*. Omenjeni modifikatorji so podobni tistim v drugih objektno orientiranih programskih jezikih, razen za modifikator *external*. Takšna external funkcija se lahko kliče samo iz zunanjih pametnih pogodb s pomočjo transakcij. Poleg omenjenih modifikatorejv dostopa Solidity pozna tudi modifikatorje funkcij, in sicer:

- pure: funkcije z modifikatorjem *pure* ne morejo brati ali spreminjati stanja (*state*) funkcije, to pomeni niti spremenljivk, preslikav, polj in drugo, vendar se lahko uporablja s funkcijo *revert()* in *require()*,
- view: v funkciji view se podobno kot pri uporabi modifikatorja *pure* ne morejo spremeniti nobeni parametri ali stanje (*state*) funkcije, vendar lahko funkcija to stanje prebere,
- payable: kadar želimo, da funkcija prejme Ether, jo moramo opremiti z modifikatorejm *payable*. Omenjeni modifikator nam omogoča vse vrste osnovnega rokovanja s sredstvi (transfer, nakazilo, hranjenje).

3.1.6 Napake prekoračitve (angl. Overflow errors)

V Solidity lahko vse operacije s celimi števili povzročijo napako prekoračitve spominskega prostora. Pravzaprav v Solidity ni nobena operacija odporna na tovrstno napako, kar pomeni, da je treba vse operacije preveriti ročno. S prevajalnikom od različice 0.8.0 naprej je sedaj mogoče preveriti aritmetične operatorje, ki ob zaznani napaki prekoračitve na to opozori, vendar problem obstaja pri nižjih različicah Solidity, ki jih nekateri razvijalci še vedno uporabljajo.

3.1.7 Optimizacija

Pri kodiranju pametnih pogodb v Solidity je treba biti pozoren na veliko več podrobnosti, kot recimo v Javascriptu ali Javi. Vsaka transakcija pomeni klic določene funkcije, ki sproži določeno operacijo, za le-te je treba plačati tako imenovani gas, saj se vse operacije zapisujejo v verigo blokov. Gas je drug izraz za provizijo, tj. določen znesek provizije, ki se plača v Ethru. Če v pametni pogodbi ni dovolj sredstev za plačilo provizije ali je znesek provizije v funkciji nastavljen prenizko (glede na trenutne zahteve Ethereum omrežja), se vse funkcije in ves *state* povrnejo v prejšnje stanje. V drugih programskih jezikih se, če funkcija kakor koli ni optimizirana, kot posledica koda le izvaja dlje časa. Na tem mestu je prav tako treba omeniti, da je zaželeno pisanje programske kode, ki je čim krajša in čim bolj optimizirana v smislu porabe gasa. To pomeni, da bomo recimo kar se da pogosto uporabili funkcijo tipa *view*, saj ta ne spreminja stanja in je tako tudi manj cenovno potratna.

3.1.8 Podatkovne strukture in abstraktni tipi podatkov

Medtem ko imajo številni programski jeziki implementiranih več različnih podatkovnih struktur in abstraktnih tipov podatkov, je teh pri Solidity manj. Prav tako je v nekaterih primerih njihova raba omejena. Preslikava v Soidity ima lahko samo podatkovno lokacijo za shranjevanje. To pomeni, da se lahko uporablja za spremenljivke, kot referenčne vrste pomnilnika v funkcijah in kot parametre za funkcije knjižnic. Preslikav ni moč uporabiti pri

parametrih ali vrnjenih parametrih funkcij pametnih pogodb, ki so javno vidne (angl. *Public*). Te omejitve veljajo tudi za polja in strukture, ki vsebujejo preslikave.

3.1.9 Byte in bytes

Ključna beseda *bytes* je v Solidity rezervirana za dinamičen seznam *byte[]*, ki zasede v pomnilniku velikost glede na število elementov v njem. Za seznam statične velikosti se v Solidity uporablja vrsta spremenljivke *bytes1*, *bytes2*, *bytes3* … *bytes32*, pri čemer v poimenovanju bytesN sam N predstavlja ***število bitov za shranjevanje v pomnilniku. Spremenljivk tipa bytes v Solidity med seboj ni moč*** neposredno primerjati. Da bi spremenljivki vseeno lahko med seboj primerjali, je treba vsako pretvoriti v hash s pomočjo keccak256 zgostitvene funkcije in potem te vrednosti med seboj primerjati.

3.2 Decentralizirane programske aplikacije

Decentralizirane aplikacije (*Dapp*) se v teoriji pogosto omenjajo kot aplikacije P2P (*peer-to-peer*) brez enega samega centralnega strežnika ali nadzorne entitete, kot je to nasprotno značilno za aplikacije, razvite po modelu odjemalec-strežnik. Razlika med decentraliziranimi in klasičnimi aplikacijami je v tem, da decentralizirane aplikacije za hranjenje programske logike in podatkov uporabljajo tehnologijo verig blokov, medtem ko klasične aplikacije za te namene uporabljajo strežnik (Metcalfe, 2020). Pri Dapp je hranjenje logike in podatkov v verigi blokov implementirano s pomočjo pametnih pogodb. V praksi večina decentraliziranih aplikacij za gostovanje uporabniških vmesnikov še vedno uporablja iste rešitve, kot to počnejo klasične aplikacije, tj. strežnik. Vendar že danes obstaja tehnologija za decentralizirano hrambo datotek, temelječa prav tako na verigi blokov. Gre za več različnih projektov, in sicer IPFS, SIA in Filecoin. Prednost uporabe decentraliziranih aplikacij je predvsem v tem, da lahko uporabnik, preden izvede določeno interakcijo z aplikacijo, preveri, kako bo aplikacija delovala, saj je izvorna koda javno dostopna vsem.

4 VARNOST PAMETNIH POGODB

Ker so Ethereum in druge platforme verig blokov še relativno nove, medtem ko je njihova tehnologija v veliki večini še v eksperimentalni fazi, so spremembe in nadgradnje programske kode relativno pogoste. Razvijalci pametnih pogodb zato lahko pričakujejo nenehen pojav novih varnostnih rešitev in razvoja dobrih praks, saj je pojav novih hroščev in varnostnih tveganj v primerjavi z drugimi, bolj etabliranimi računalniškimi platformami, pogostejši.

V nadaljevanju smo zbrali različne hrošče in varnostne pomanjkljivosti programskega jezika Solidity, ki smo jih našli v literaturi in na spletu. Le-te smo razdelili v pet kategorij:

- ranljivost funkcij in metod,
- ranljivost spremenljivk,
- ranljivost v povezavi s klici,
- ranljivosti s povezavi z gasom,
- druge ranljivosti.

Z varnostnimi pomanjkljivostmi programskega jezika Solidity so se v svojih prispevkih ukvarjali že številni avtorji, kot recimo Riady (Riady, 2018), Chen, Manning in drugi. Nekatere varnostne pomanjkljivosti so zaradi hekerskih napadov v preteklosti v veliki večini tudi že splošno znane in jih zato omenjajo v raznih anonimnih seznamih, na forumih in v nestrokovnih člankih. V naslednjih poglavjih smo različne varnostne pomanjkljivosti, ki smo jih zbrali s pomočjo kvantitativne raziskovalne metode, in sicer s pomočjo pregleda tako strokovne literature kot nestrokovnih prispevkov na spletnih straneh, preučili in jih razvrstili glede na področje ranljivosti. Kjer je bilo to mogoče, smo navedli primer ranljivosti tudi s prikazom v izvorni kodi in predlagali popravke.

4.1 Ranljivosti funkcij in metod

V tem poglavju bomo našteli in opisali najpogostejše ranljivosti funkcij in metod v programskem jeziku Solidity.

4.1.1 Privzeta vidnost funkcij (angl. Function Default Visibility)

Funkcije, ki jim ne določimo tipa vidnosti, so privzeto javne (angl. *Public*). Tip vidnosti funkcije določa, ali lahko funkcijo kličejo druge funkcije, funkcije drugih pogodb ali zunanji uporabniki (Riady, 2018). Ranljivost privzete vidnosti lahko predstavimo na naslednjem primeru:

```
contract SrecniNaslovZadaneEther {
      function nakaziNagrado() {
      require(uint16(msg.sender) == 0);
      _posljiZnesek();
      }
      function _posljiZnesek() private {
            msg.sender.transfer(this.balance);
      }
}
```

Slika 5: Ranljivost privzete vidnosti funkcij

Vir: Lasten

V prikazanem primeru lahko funkcijo _posljiZnesek pokličemo iz pametne pogodbe SrecniNaslovZadaneEther ali pogodbe, ki deduje od navedene pogodbe. Do zneska Ethrov v pametni pogodbi je tako upravičen uporabnik, ki poseduje naslov, katerega zadnja štiri heksadecimalna števila predstavljajo število nič. Nabor sodelujočih uporabnikov bi v nadaljevanju lahko recimo določili v dedovani pametni pogodbi. Brez določene vidnosti funkcije *private* bi funkcijo _posljiZnesek lahko poklical s katerega koli naslova tudi iz pametne pogodbe SrecniNaslovZadeneEther izpeljanih pametnih pogodb. Če vidnost funkcije ni določena, se funkciji dodeli privzeta vidnost, ki je *internal.*

Varnostno priporočilo:

Ustrezno vidnost funkcije je treba v Solidity vedno zapisati. Odsvetuje se, da se vidnost funkcije ne določi, saj se v tem primeru funkciji samodejno dodeli privzeta voidnost *internal*.

4.1.2 Nezaščiten klic funkcije za samouničenje (angl. Unprotected Selfdestruct Instruction)

Pri nezaščitenem klicu funkcije za samouničenje lahko potencialni napadalec zaradi pomanjkljivega nadzora dostopa sproži samouničenje pogodbe. Funkcija za samouničenje (angl. *Self destruct*) je namenjena brisanju pametne pogodbe z verige blokov, medtem ko se znesek kripto valute, ki jo hrani pametna pogodba, prenese na navedeni naslov (Chen, Xia, Lo in Grundy, 2021). Primer neustrezno zaščitenega dostopa:

```
contract Zrtev {
      /…/
            function unicenje() public {
                  selfdestruct(msg.sender)
      }
}
```

Slika 6: Ranljivost nazaščitenega klica funkcije za samouničenje

Vir: Lasten

Varnostno priporočilo:

Ker je msg.sender zadnji naslov, ki je interagiral s pametno pogodbo, in ker msg.sender ni inicializiran na lastnika pametne pogodbe, lahko funkcijo selfdestruct() pokliče praktično kdor koli. Da bi odpravili varnostno pomanjkljivost, je treba msg.sender ustrezno inicializirati in uporabiti konstruktor:

```
contract NisemZrtev {
	address private owner
	constructor() {
		owner = msg.sender
	}
	…
	function unicenje() public {
		require(msg.sender == owner)
		selfdestruct(msg.sender)
	}
}
```

Slika 7: Odpravljena ranljivost nezaščitenega klica funkcije za samouničenje

Vir: Lasten

4.1.3 Uporaba zastarelih funkcij (angl. Use of Deprecated Solidity Functions)

Funkcije in metode prejšnjih različic Solidity sčasoma postajajo zastarele, prav tako se v kombinaciji z novejšimi funkcijami v prevajalniku pojavljajo napake ali nepredvideni rezultati. Zastarele so recimo funkcije in metode: *suicide(address), sha3(...), callcode(...), throw(...), msg.gas, constant*.

Varnostno priporočilo:

Navedene funkcije je priporočljivo nadomestiti s *selfdistruct(address), keccak256(...), delegatecall(...), revert(), gasleft* in *pure* ter *view*. Prav tako pri inicializaciji spremenljivk ni več priporočeno uporabljati predpone *var,* temveč jo vsakič nadomestiti z ustrezno oznako za vrsto spremenljivke.

4.1.4 Nepravilno poimenovanje konstruktorja (angl. Incorrect Constructor Name)

V Solidity je konstruktor še posebej pomemben, saj v njem nastavimo lastnika pametne pogodbe, iz česar za njegov naslov izhajajo določene pravice, ki jih naj ne bi imel nihče drug. V primeru napačnega klica konstruktorja, recimo njegovega napačnega poimenovanja, do klica

konstruktorja ne pride, temveč se izvede običajna funkcija. Posledica tega je lahko, da pametna pogodba nima nastavljenega lastnika pogodbe, kar pomeni, da se lahko kot lastnik pogodbe inicializira kdor koli.

Od različice Solidity 0.4.22 naprej konstruktorja več ni treba poimenovati. Po novem se za rezerviranim poimenovanjem *constructor* v oklepajih napiše le telo konstruktorske funkcije. To pomeni, da možnost napačnega poimenovanja konstruktorja od navedene različice 0.4.22 naprej ne predstavlja več nobene težave.

V spodnjem primeru smo konstruktor namenoma napačno formulirali. Tako ne pride do njegovega klica, ampak se *JazSemKonstruktor* pokliče kot običajna funkcija. Konstruktor bi namreč do različice 0.4.22 moral biti poimenovan enako kot pametna pogodba, tj. v našem primeru *Konstruktor*, in ne *JazSemKonstruktor* ali kakor koli drugače.

```
contract Konstruktor {
/…/
       function JazSemKonstruktor()
         public
    {
        owner = msg.sender;
    }
}
```

Slika 8: Ranljivost nepravilnega poimenovanja konstruktorja

Vir: Lasten

```
contract Konstruktor {
/…/
       constructor()
         public
    {
        owner = msg.sender;
    }
}
```

Slika 9: Odpravljena ranljivost nepravilnega poimenovanja konstruktorja

Vir: Lasten

Varnostno priporočilo:

Tudi z vidika varnosti je smotrno vedno uporabljati novejše različice programskega jezika Solidity, ali vsaj tiste, ki sledijo večjim varnostnim popravkom. V konkretnem primeru se priporoča uporaba različice 0.4.22 ali višje.

4.1.5 Nepravilen vrstni red dedovanja (angl. Incorrect Inheritance Order)

Solidity podpira večkratno dedovanje, kar pomeni, da lahko ena pametna pogodba deduje od več drugih pametnih pogodb. Vendar večkratno dedovanja vnaša dvoumnost. Namreč, če dve ali več osnovnih pametnih pogodb opredeli isto poimenovano funkcijo, v programskem jeziku Solidity ni opredeljeno, katera osnovna pogodba ima prioriteto.

Varnostno priporočilo:

Rešitev je v dobrem razumevanju in dosledni uporabi C3-pravil linearizacije. Spodnji primer ponazarja večstopenjsko dedovanje. Zapisana programska koda spodnjega primera bo producirala naslednji vrstni red klicev konstruktorjev: C, B, A. Zato je treba biti pri določitvi vrstnega reda dedovanja skrben. Priporoča se tudi uporaba avtomatiziranih testov in uporaba po nasvetu bolj izkušenih programerjev.

```
contract A {
    constructor() public {}
}

contract B {
    constructor() public {}
}

contract C is A, B {
    constructor() public {}
}
```

Slika 10: Večstopenjsko dedovanje

Vir: Lasten

4.1.6 Nezaščiten dvig Ethra (angl. Unprotected Ether Withdrawal)

Pri nezaščitenem dvigu Ethra gre predvsem za neustrezno zaščito dostopa oz. dovoljenj, kdo lahko interagira s pametno pogodbo in katere funkcije lahko pokliče (Müller, 2018). To pomeni, da pametna pogodba preprosto ni imela dovolj visoke stopnje zaščite, da bi napadalcu onemogočila klice kritičnih funkcij. Za skoraj vsako interakcijo s pametno pogodbo (tj. vse funkcije, ki spreminjajo stanje pametne pogodbe) je potrebna transakcija kriptovalute. Za varnostno ranljivost so sicer običajno dovzetne funkcije tipa *payable* in metode, povezane z določanjem pravic glede interakcij s pametno pogodbo.

Varnostno priporočilo:

Naslednji primer pogodbe ima ustrezno zaščito funkcije z modifikatorjem.

```
contract seznamDovoljenihNaslovov {

	// konstruktor

	modifier onlyOwner() {
		require(clani[msg.sender] != 0);
		_;
	}

	mapping (address => bool) clani;
	constructor() public Ownable() {
	}
	function dodajClana(address _clan) public onlyOwner {
			clani[_clan] = true;
	}
}
```

Slika 11: Odpravljena ranljivost nezaščitenega dviga Ethra

Vir: Lasten

Navedeni primer prikazuje pametno pogodbo, s katero lahko lastnik pogodbe doda na seznam druge uporabnike. Ko se konstruktor zažene (to se zgodi samo enkrat, ob objavi pametne pogodbe na verigi blokov), tip konstruktorja *Ownable* ob objavi pametne pogodbe naslov uporabnika, s katerega je bila pogodba objavljena, nastavi za lastnika pametne pogodbe. Pri tem modifikator *onlyOwner* v funkciji *dodajClana* povrne stanje (angl. *State*) v predhodno stanje, če funkcije ni poklical lastnik pametne pogodbe.

4.2 Ranljivosti spremenljivk

V tem poglavju bomo zbrali in podrobno opisali ranljivosti, povezani z uporabo spremenljivk v programskem jeziku Solidity.

4.2.1 Privzeta vidnost spremenljivk stanja (angl. State Variable Default Visibility)

Z določanjem vidnosti spremenljivk določamo, kdo lahko dostopa do posamezne spremenljivke. Solidity pozna deklaracije *public*, *internal* in *private*. Primer je zelo podoben predhodno opisani pomanjkljivosti v povezavi s privzeto vidnostjo funkcij.

4.2.2 Napaka prekoračitve in podkoračitve celih števil (angl. Integer Overflow and Underflow)

Napaka prekoračitve in podkoračitve se zgodi, ko pri aritmetični operaciji spremenljivka doseže največjo ali najnižjo predpisano vrednost. Če je recimo neko število shranjeno v tipu spremenljivke uint8 (cela števila od 0 do 255), se napaka prekoračitve in podkoračitve zgodi, če določena operacija želi ustvariti numerično vrednost izven tega okvirja, torej –1 ali 256., ki je ne doseže, temveč se štetje začne ponovno od števila nič. Pomanjkljivost je še ta, da se ne

pojavi nobeno opozorilo o napaki. Za primer funkcije, ki je ranljiva za napako prekoračitve in podkoračitve lahko navedemo funkcijo *povecajCasOmejitve*. Ta funkcija je namenjena povečanju časovne omejitve, kdaj lahko posameznik nakaže kripto valuto iz pametne pogodbe na svoj račun. Pri napaki prekoračitve lahko napadalec na ta čas vpliva, kljub temu da je minimalni čas nastavljen na en teden. Napad bi lahko izvedli s pomočjo druge pametne pogodbe, pri objavi katere bi navedli naslov, iz katerega je objavljena pametna pogodba *Zrtev*. V novi pametni pogodbi bi opredelili funkcijo, s katero bi povečali število sekund za vrednost, ki presega *uint* število.

```
contract Zrtev {
	/…/
	function nakazilo() external payable {
		stanje[msg.sender] += msg.value;
		casOmejitve[msg.sender] = now + 1 weeks;
	}
	function povecajCasOmejitve(uint _sekundePodaljsanja) public {
		casOmejitve[msg.sender] += _sekundePodaljsanja;
	}
	/…/
}
```

Slika 12: Ranljivost napake prekoračitve in podkoračitve

Vir: Lasten

Varnostno priporočilo:

Za zaščito napake prekoračitve in podkoračitve je zato priporočljivo uporabiti funkcijo SafeMath za uint števila iz knjižnice OpenZeppelin. Knjižnico v pogodbo najprej uvozimo, nato v prvi vrstici pogodbe uporabimo stavek *using SafeMath for uint*. Funkcija *povecajCasOmejitve* bi nato bila videti takole:

```
using SafeMath for uint;

function povecajCasOmejitve(uint _sekundePodaljsanja) public {
	casOmejitve[msg.sender] = casOmejitve
	[msg.sender].add(_sekundePodaljsanja );
}
```

Slika 13: Odpravljena ranljivost napake prekoračitve in podkoračitve

Vir: Lasten

Pri tem metoda *add* poskrbi za preprečevanje ranljivosti prekoračitve in podkoračitve. Če bi število *_sekundePodaljsanja* povzročilo prekoračitev ali podkoračitev, bi ta funkcija vrnila napako, medtem ko se funkcija *povecajCasOmejitve* se ne bi izvedla.

4.2.3 Neinicializiran kazalec v pomnilniku (angl. Uninitialized Storage Pointer)

Neinicializirane lokalne spremenljivke lahko kažejo na nepredvideno lokacijo v pomnilniku pametne pogodbe, kar lahko povzroči ranljivost. Ti kazalci bi lahko namreč nepredvideno kazali na prostor drugih spremenljivk v pomnilniku.

Varnostno priporočilo:

Napaka je bila sicer odpravljena v različici Solidity 0.5.0, zato je priporočljivo uporabljati višnje, tj. novejše različice programskega jezika. Pogodbe z neinicializiranimi pointerji v pomnilniku prevajalnik niti več ne prevede. Če bi vseeno želeli uporabiti nižjo različico jezika, je treba vse spremenljivke opremiti z atributom *memory* (Antonopoulos in Wood, 2018).

4.2.4 Avtorizacija s tx.origin (angl. Authorization through tx.origin)

Spremenljivka tx.origin je globalna spremenljivka, ki vrne naslov računa, s katerega je bila poslana transakcija. Uporaba te spremenljivke za avtorizacijske klice predstavlja veliko ranljivost.

Varnostno priporočilo:

Namesto nje je bolje uporabiti spremenljivko msg.sender, ki vrača naslov računa, s katerega se opravlja klic pametne pogodbe (Modi, 2018).

4.3 Ranljivosti v povezavi s klici

V tem poglavju bomo opisali ranljivosti v programskem jeziku Solidity v povezavi s klici funkcij in metod.

4.3.1 Nepreverjene vrnjene vrednosti klicev (angl. Unchecked Call Return Value)

Za opravljanje prenosa kriptovalute iz ene pogodbe na drugo razvijalci uporabljajo metode *transfer, call* in *send.* Vendar sta *call* in *send* funkciji tipa boolean, kar pomeni, da po izvršitvi vrneta vrednost *true* ali *false,* če vrnjena vrednost *false* nma možnosti za povrnitev v prejšnje stanje, ki je bilo pred transakcijo. Napaka se zgodi, ko razvijalec v kodi pozabi predvideti preverjanje vrnjene vrednosti. Primer je recimo pametna pogodba, podobna lotu. Zmagovalec prejme določen znesek, vendar zaradi narave transakcij veriženja blokov običajno nekaj »drobiža« ostane v denarnici oz. na naslovu pogodbe. Le-tega si potem lahko razdelijo drugi igralci. Če v pametni pogodbi za transfer sredstev uporabimo *send*, a transakcija zaradi določenih razlogov ne uspe (v zadnjem času visoke prezasedenosti omrežja, premalo gasa za plačilo transferja ob nekonsistentnih količinah gasa itd.), bo funkcija *send* vseeno vrnila vrednost true, saj je bila izvedena, kljub temu da sredstva zaradi omenjenih dejavnikov morda

sploh niso bila nakazana. Ko je vrednost *send* nastavljena na *true*, lahko zneska, ki sploh ni bil poslan, uporabljajo drugi igralci. Primer funkcije:

```
function posljiZmagovalcu() public {
    require(!nakazano);
    zmagovalec.send(balance);
    payedOut = true;
}
```

Slika 14: Ranljivost nepreverjene vrnjene rednosti klicev

Vir: Lasten

```
function posljiZmagovalcu() public {
    require(!nakazano);
    zmagovalec.transfer(balance);
    payedOut = true;
}
```

Slika 15: Odpravljena ranljivost nepreverjene vrnjene rednosti klicev

Vir: Lasten

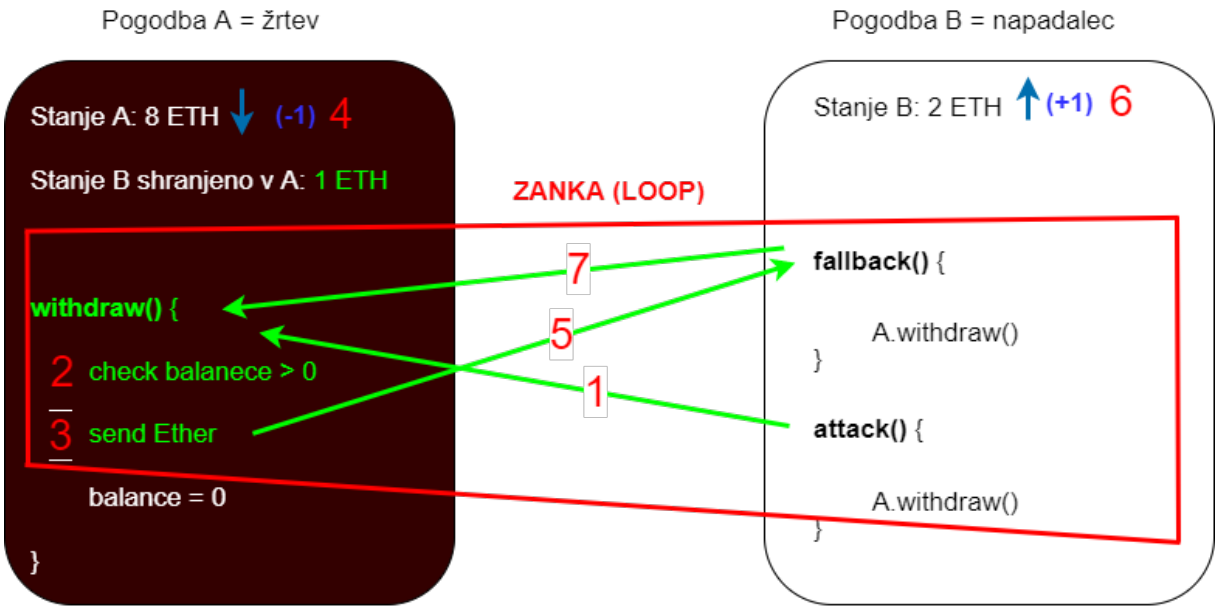
Varnostno priporočilo:

Namesto metode *send()* je priporočeno uporabljati metodo *transfer()*, ki omogoča nadzor zneska gasa (bodisi z ustreznimi metodami bodisi prek denarnice Metamask) ali metodo *call()* v kombinaciji z implementacijo ustrezne zahteve za potrditev uspešne izvedbe transakcije (glej poglavje 4.4.1) in zaščito pred ranljivostjo ponovnega vstopa (glej poglavje 4.3.2).

4.3.2 Ponovni vstop (angl. Reentrancy)

Solidity ima to lastnost, da lahko funkcije ene pametne pogodbe kličejo funkcije iz drugih pametnih pogodb. Zunanji klici funkcij in nakazila Ethra drugim pametnim pogodbam dopuščajo napadalcem priložnosti za izvedbo napada. Zunanje klice je namreč možno izrabiti in z njimi sprožiti izvajanje funkcij drugih pogodb na način, kot razvijalci tega niso predvideli. Ponovni vstop tako označuje varnostno pomanjkljivost, pri kateri napadalec v prvi interakciji s pametno pogodbo najprej sproži izvedbo neke funkcije ali metode, nato njeno izvajanje izkoristi in pokliče neko drugo metodo na način, kot to ni bilo predvideno ali v času, ko napadena pogodba še izvaja določene poklicane operacije. Primer takšnega napada bi bili dve pogodbi. Pogodba A hrani sredstva raznih uporabnikov in drugih pogodb. Pogodba A ima funkcijo, ki omogoča dvig denarja uporabnikom, ki so ji poslali svoja sredstva. Funkcija za nakazilo je sestavljena iz treh metod, in sicer preverjanja stanja ETH za konkretnega uporabnika (če je

večje od 0), transakcije ETH in na koncu posodobitve stanja. Pogodba B je pogodbi A poslala 1 ETH. Napadalec iz pogodbe B lahko z metodo iz ene funkcije pokliče funkcijo za nakazilo iz pogodbe A. Ta preveri stanje, ki je večje od nič in pošlje sredstva pogodbi B. Ko ta prispejo, se v pogodbi B sproži funkcija *fallback()*, ki ima prav tako metodo za klic nakazila pogodbe A. Ker se stanje v A preverja na koncu, le-to za stanje sredstev pogodbe B v pogodbi A ne kaže nič in funkcija iz B lahko kliče funkcijo iz A, vse, dokler v pogodbi A ne bo zmanjkalo sredstev (ETH).



Slika 16: Shema poteka napada pri ranljivosti ponovnega vstopa

Vir: Lasten

V nadaljevanju sta navedena primer ranljive pogodbe *contract A* in primer popravljene ranljivosti v pametni pogodbi *contract Apopravljen*.

```
contract A {
	/…/
	function nakazilo(uint _znesek) public {
		require(balances[msg.sender] >= _znesek;
		(bool, sent) = msg.sender.call{value: _znesek} ('');
		balances[msg.sender] = _amount
	}
	/…/
}

contract APopravljen {
	/…/
	bool internal locked;
	modifier noReentrant() {
		require(!locked, 'Ta pogodba onemogoča ponovni vstop';
		locked = true:
		;
		locked = false;

	}
	function nakazilo(uint _znesek) public noReentrant {
		require(balances[msg.sender] >= _znesek;

		balances[msg.sender] = _amount

		(bool, sent) = msg.sender.call{value: _znesek} ('');
		require(sent, 'Pošiljanje ETH ni uspelo')

	}
	/…/
}
```

Slika 17: Popravljena ranljivost ponovnega vstopa

Vir: Lasten

Varnostno priporočilo:

Napad na ponovni vstop bi v tem primeru lahko preprečili, če bi, kot smo navedli v zgornjem primeru, stanje na računu oz. vrednost spremenljivke *balances* posodobili, preden bi dovolili izvedbo nadaljnjih operacij. V pomoč je prav tako modifikator *noReentrant*, ki zaklene pogodbo za čas, dokler se izvaja katera koli funkcija v pogodbi. Tako se istočasno lahko izvaja le ena funkcija.

4.3.3 Delegiran klic na nepreverjeno pogodbo (angl. Delegatecall to Untrusted Callee)

Delegirani klic je klic funkcije, s pomočjo katere lahko primarna pametna pogodba dinamično in v času izvajanja naloži programsko kodo druge pametne pogodbe, pri tem *msg.sender* in *msg.value* metodi ne spreminjata svojih vrednosti. Funkcionalnost je namenjena predvsem klicanju funkcij posameznih knjižnic. Ob tem moramo biti kot razvijalci pozorni na to, da *delegatecall* ohranja kontekst funkcij ene in druge pogodbe ter da naslovi *storage, balance* in

drugi naslovi še vedno pripadajo pogodbi, iz katere izvajamo *delegatecall*. Pri tem se lahko spremenijo vrednosti spremenljivk oz. sloti v pomnilniku. Delegirani klici v druge, predvsem nepreverjene pametne pogodbe, so nevarno početje ravno zato, ker lahko koda iz klicanega naslova spreminja kodo klicatelja, predvsem vrednosti spremenljivk. Tako lahko pride tudi do spremembe vrednosti spremenljivke primarne pogodbe *balances*. Pogodba, od katere se izvede klic, ima tako lahko popolni nadzor nad klicano pogodbo.

Varnostno priporočilo:

Zaradi zgoraj navedenega morajo biti spremenljivke stanja obeh pogodb opredeljene v enakem zaporedju, prav tako morajo biti tipi spremenljivk isti. Vsekakor je priporočljivo, da se kot razvijalci kar se da izogibamo uporabi klicev *delegatecall*, če jih že moramo uporabljati, zgolj s pogodbami, ki so preverjene (Foster, 2018) in uporabljajmo knjižnice, ki ne spreminjajo stanja (angl. *Stateless*) (Manning, 2018).

4.4 Ranljivosti v povezavi z gasom

V tem poglavju bomo zbrali in opisali ranljivosti pametnih pogodb, povezanih z enoto gas.

4.4.1 Nezadostna koičina gasa (angl. Insufficient Gas Griefing)

Napadalec bi lahko v določeni pogodbi, ki sprejme generične podatke in jih s funkcijo *address.call()*uporablja pri subsidiarnih klicih v druge pametne pogodbe, vplival na to, kateri klici bodo izvedeni in kateri ne. Če subsidiarni klic ne uspe zaradi premajhne količine gasa, ki je na voljo za plačilo transakcije (tj. posamezne operacije), lahko pogodba, ki izvede subsidiarne klice, le-te cenzurira. Primer bi bila pogodba o glasovanju za določeno odločitev (volilne aplikacije, referendumi). Napadalec sicer ne bi imel čiste posredne koristi v obliki kriptovalute, vendar bi lahko cenzuriral glasove, ki mu ne bi bili po volji, tako da bi povzročil, da določene transakcije zaradi prenizke količine gasa ne bi uspele, običajno uporabniki tovrstnih aplikacij ne morejo glasovati dvakrat.

Varnostno priporočilo:

Da se omenjenemu izognemo, je treba biti pozoren, da imajo pogodbe, ki jih kličemo s subsidiarnimi klici, možnost, da uporabnik sam določi minimalni znesek za plačilo gasa. V spodnjem primeru vidimo najprej ranljivo funkcijo v pametni pogodbi.

```
function ranljivaFunkcija(ZunanjaPogodba target, bytes memory _data) public
returns(bool) {
      (bool, success, ) =
address(target).call(abi.encodeWithSignature('execute(bytes)', _data));
      return success;
}
```

Slika 18: Ranljivost nezadostne količine gasa

Vir: Lasten

Funkcija *ranljivaFunkcija* kliče funkcijo *execute()* v drugi pametni pogodbi *ZunanjaPogodba*. Ker funkcija *ranljivaFunkcija* ne opravi nobenega preverjanja, koliko gasa je še na voljo ali kakšen je rezultat klica funkcije v pametni pogodbi *ZunanjaPogodba*, se le-ta izvede brez napak.

```
uint public predvidenZnesekGasa = 1000000;
uint public gasPotrebenMedKlici = 10000;

function zdravaFunkcija(ZunanjaPogodba target, bytes memory _data) public
returns(bool) {
      uint256 gasNaVoljo = gasleft() - gasPotrebenMedKlici;
      require(gasNaVoljo - gasNaVoljo / 64 >= predvidenZnesekGasa, 'Na
voljo ni dovolj gasa');
      (bool, success, ) =
address(target).call(abi.encodeWithSignature('execute(bytes)', _data,
predvidenZnesekGasa));
      return success;
}
```

Slika 19: Popravljena ranljivost nezadostne količine gasa

Vir: Lasten

Spremenljivka *predvidenZnesekGasa* predstavlja znesek gasa, ki ga potrebuje metoda *call()* za pravilno izvedbo funkcije *execute()* iz pametne pogodbe *ZunanjaPogodba*. Spremenljivka *gasNaVoljo* predstavlja 1/64 zneska gasa, ki bo ostal v pogodbi po klicu metode *call()*. Število 64 predstavlja privzeto vrednost 1/64, določeno v Ethereum Virtual Machine, ki po klicu iz druge pogodbe ostane v klicani pametni pogodbi. Najprej izračunamo znesek gasa, ki bo na voljo pred izvedbo metode *call()*. Nato preverimo, ali bo v pametni pogodbi preostali gas po klicu večji od *predvidenZnesekGasa* in če je odgovor da, se funkcija izvede.

4.4.2 Fiksno kodiran znesek gasa funkcij (angl. Message call with hardcoded gas amount)

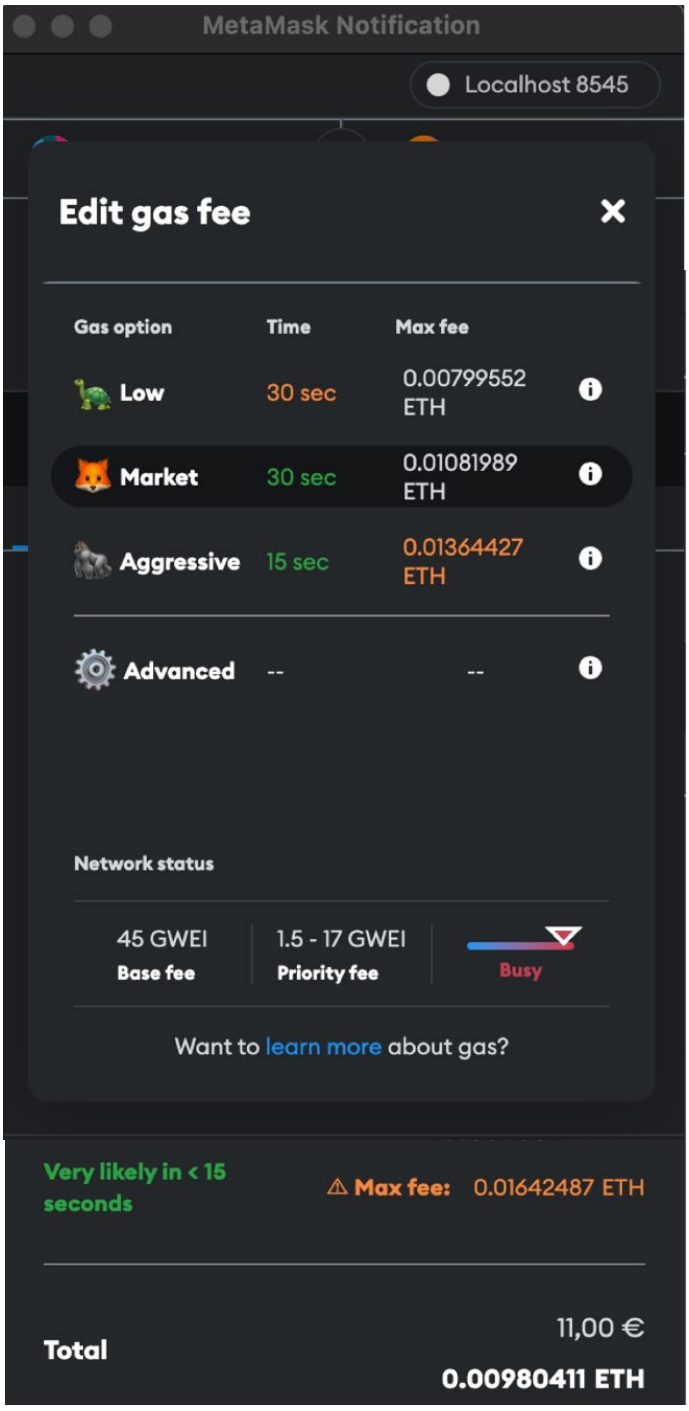
Funkciji *transfer()* in *send()* imata fiksno kodirano količino gasa, ki ga porabita ob svoji izvedbi (tj. 2300 gwei, kar predstavlja 0.00000230 ETH). Ko je Ethereum omrežje zelo zasedeno, se poveča tudi količina potrebnega gasa za izvedbo transakcije. Ta lahko krepko preseže fiksno

kodiran znesek gasa. V poglavju 4.3 smo pri opisu ranljivosti *Nepreverjene vrnjene vrednosti klicev* sicer zapisali, da je funkcija *transfer* varnejša, ker ne vrne logične vrednosti in jo je zato težje manipulirati. V luči vedno višjih cen za plačilo transakcij to pomeni, da se lahko zgodi, da v času, ko je omrežje zelo zasedeno, pametne pogodbe, ki uporabljajo funkcije s fiksno kodiranim zneskom gasa, kot ste *transfer* in *send,* ne bodo izvedle.

Varnostno priporočilo:

S posodobitvijo omrežja Eteherum, imenovano EIP-1550 (London upgrade), ki se je zgodila 5. 8. 2021, je bil uveden boljši mehanizem predvidevanja cene gasa za opravljeno transakcijo. Vendar London upgrade ni povsem odpravil težave, zato je v danem primeru včasih celo bolje uporabiti manj varno funkcijo *call()*z implementacijo mehanizma, ki preveri, ali se je funkcija pravilno izvedla ter poleg tega dodatno implementirati tudi ustrezno zaščito pred ranljivostjo ponovnega vstopa (glej poglavje 4.3.2). Druga in še nekoliko varnejša možnost na strani uporabnika aplikacije, je denarnica Metamask, saj lahko uporabnik v njej ob izvedbi želene transakcije med ponujenimi zneski gasa, preprosto izbere tistega z višjim zneskom. Spodnja dva posnetka denarnice Metamask prikazujeta izbiro višine zneska gasa za izvedbo transakcije.

Slika 20 prikazuje možnosti, ki jih ima uporabnik na voljo za izbiro višine zneska gasa. Zraven zneskov je naveden tudi čas, ki bo predvidoma potreben za izvedbo transakcije.



Slika 20: Metamask (vse možnosti izbire zneska gasa)

Vir: Lasten

4.4.3 DoS-napad s pomočjo omejitve gasa bloka (angl. DoS With Block Gas Limit)

Opis:

Vsak blok v verigi blokov ima omejeno količino podatkov in s tem tudi omejeno količino gasa. Če imamo v pametni pogodbi polje znakov, recimo z vsemi naslovi, ki jim želimo nakazati neki znesek in je to polje preveliko v smislu, da število naslovov presega velikost bloka, transakcija ne bo uspela. Prav tako bi naše polje znakov lahko manipuliral napadalec s pomočjo DoS-napada.

Varnostno priporočilo:

Zato je priporočljivo, da če v pametnih pogodbah izvajamo *for zanke* na poljih z neopredeljeno velikostjo, transakcije razbijemo v več različnih transakcij, ki bodo potrjene v več blokih.

4.5 Druge ranljivosti

4.5.1 Določanje časa iz časovnih vrednosti blokov (angl. Block values as a proxy for time)

Opis:

Uporaba metod, ki vračajo vrednosti iz verige blokov (recimo čas bloka oz. *Block time*) v funkcijah pametne pogodbe in jih uporabljajo v drugih funkcijah, ni priporočljiva. Težava je v tem, da zaradi razlik v sinhronizaciji omrežja blokov te vrnjene vrednosti niso vedno v vseh primerih in za vse uporabnike identične. Zato niso vedno varne za uporabo. Metoda *block.timestamp* se morda zdi primerna, da nanjo vežemo časovno relevantne operacije, vendar temu ni tako. Ethereum omrežje je zelo decentralizirano, zato lahko vozlišča čas sinhronizirajo samo do določene mere. Podobna težava je z metodo *block.number*.

Varnostno priporočilo:

Zato je priporočljivo, da v takšnem primeru podatke pridobimo iz omrežja decentraliziranih računalnikov, ki zagotavlja zanesljive podatke iz resničnega sveta za pametne pogodbe, kot je recimo omrežje Chainlink.

V nadaljevanju smo v okviru tega diplomskega dela pripravili kontrolni seznam za preverjanje upoštevanja varnostnih priporočil za Solidity razvijalce.

4.6 Kontrolni seznam

V predhodnih poglavjih opredeljene varnostne ranljivosti smo skušali združiti v po abecednem vrstnem redu oblikovan kontrolni seznam, ki bi nam naj pri pisanju empiričnega dela tega diplomskega dela bil v pomoč pri spoštovanju varnostnih priporočil. Seznam ima tudi navedene

kategorije oz. področja, na katera se varnostna pomanjkljivost nanaša, tudi navedbo priporočenih in odsvetovanih praks ter tudi kratek primer izvorne kode, kjer je bilo to mogoče

Tabela 1: Kontrolni seznam za preverjanje ranljivosti pametnih pogodb

	KATEGORIJA	RANLJIVOST	ODSVETOVANO	PRIPOROČLJIVO	PRIMER
1	spremenljivke	**Avtorizacija s tx.origin**	tx.origin	msg.sender	owner = msg.sender
2	drugo	**Določanje časa iz vrednosti blokov**	block.timestamp, block.number	uporaba podatkov iz omrežja Chainlink	Import Chainlinkovega Aggregator iterface-a in uporaba funkcije getTimestamp()
3	gas	**DoS-napad z omejitvijo gasa bloka**	zanka s spremenljivko, na katere vrednost lahko vpliva napdadalec/uporabnik; funkcijo call je treba uporabljati s previdnostjo, ker privzeto ves preostali gas nakaže prejemniku zneska iz pametne pogodbe	uporaba nekompleksnih zank, pri katerih ne sme število iteracij preseči gasa, ki je na volj za blok uporabo Pull over Push vzorca programiranja	(bool send,) = prejemnik.call{value: znesek} pretvorimo v function widthdraw public ()
4	gas	**Fiksno kodiran znesek gasa funkcij**	uporaba funkcij *transfer* in *send*, ki imata fiksno kodiran znesek gasa (2300 gwei)	uporaba klica call z zaščito pred ponovnim vstopom (angl. *Reentrancy*)	.call.value(...)(»«)

5	klici	**Klic *delegatecall***	uporaba delegatecall ni priporočljiva na nepreverjene naslove (pogodbe) ali skupaj s spremenljivkami, ki jih vnesejo uporabniki: require(callee.delegatecall(_data));	uporaba delegatecall samo na preverjene naslove, tj. pogodbe in kadar je to res nujno potrebno	contract VarenDelCall {address callee;
					address owner;
					
					modifier onlyOwner {
					require(msg.sender == owner);
					}
					constructor() public {
					callee = address(0x0);
					owner = msg.sender;
					}

					function setCallee(address newCallee) public onlyOwner { callee = newCallee; } function forward(bytes _data) public { require(callee.delegatecall(_data)); } }
6	spremenljivke	**Napaka prekoračitve in podkoračitve**	poraba brez SafeMath	Uporaba SafeMath knjižnice Open Zeppelin	using SafeMath for uint;
7	spremenljivke	**Neinicializiran kazalec pomnilniku**	uint stateVariable;	uint stateVariable = 0;	contract Pogodba { uint x; function Funkcija (uint x = 0;) }
8	funkcije	**Nepravilen vrstni red dedovanja**	contract Three is One, Two	contract Three is Two, One	vrstni red dedovanja sledi od desne proti levi, tj. Three ← Two ← One
9	funkcije	**Nepravilno poimenovanje konstruktorja**	uporaba različic Solidity, nižjih od 0.4.22, ni priporočljiva, saj je v njih konstruktor moral biti poimenovan z istim imenom kot pametna pogodba	uporabljamo različico Solidity, višjo od 0.4.22	constructor() public { owner = msg.sender; }
10	klici	**Nepreverjene vrnjene vrednosti klicev**	uporaba funkcije *send* brez preverjanja vrnjene (*return*) *vrednosti*	uporaba funkcije *transfer* ali funkcije *call* in peverjanje vrnjene vrednosti ** glej pomanjkljivost pri točki 4!	msg.sender.call(this.balance);

11	gas	**Nezadostna količina gasa**	funkcija nima opredeljene minimalne količine gasa za izvedbo vseh funkcij	zahtevati, da za izvršitev funkcije in vseh ostalih funkcij klicatelj zagotovi dovolj gasa	function dovoljGasa(bytes _data, uint _gasLimit) {
					require(gasleft() >= _gasLimit);
					...
					}

12	funkcije	**Nezaščiten klic funkcije za samouničenje**	msg.sender ni inicializiran na lastnika	msg.sender mora biti obvezno inicializiran na lastnika pogodbe, v nasprotnem primeru se kot msg.sender smatra zadnji naslov, ki je interagiral s pogodbo	address private owner
					constructor() {
					owner = msg.sender
					}

13	funkcije	**Nezaščiteno nakazilo Ethra**	funkcija nima zaščite dostopa	implementirati zaščito dostopa na način, da lahko funkcije kličejo le avtorizirani uporabniki	uporaba modulatorjev (angl. *Modyfier*):
					modifier **onlyowner** {
					if (msg.sender == creator);}
					function nakazilo()
					onlyowner { ... }

14	klici	**Ponovni vstop**	uporaba funkcij brez zaklepanja (mutex)	uporaba zaklepanja ali *mutexa*, lahko tudi s pomočjo funkcije ReentrancyGuard iz OppenZeppelin knjižnice	function withdraw() public nonReentrant { }
15	funkcije	**Privzeta vidnost funkcij**	vidnost funkcije ni podana	Podana je vidnost funkcije, recimo ***private***	function ImeFunkcije () ***private*** { }
16	spremenljivke	**Privzeta vidnost spremenljivk stanja**	vidnost spremenljivke ni podana, zato obvelja privzeta vrednost *public*	vidnost spremenljivke je podana	uint ***private*** num
17	funkcije	**Uporaba zastarelih funkcij**	*suicide(address), sha3(), callcode(), throw(), msg.gas, constant*	*selfdistruct(address), keccak256(), delegatecall(), revert(), gasleft, pure/view*	function primer() public **view** returns(string) { }
Vir: Lasten					

4.7 Varnostne ranljivosti in rešitve, objavljene s strani organizacije Ethereum

Organizacija Ethereum varnostne ranljivosti programskega jezika Solidity na svojih spletnih straneh spremlja oz. objavlja le do neke mere, saj trenutno nima objavljenega seznama vseh obstoječih ranljivosti, prav tako so le-te objavljene skupaj s priporočili. Na svoji spletni strani organizacija Ethereum sicer navaja nekaj primerov dobrih praks, vendar bralca kljub temu za podrobnejši pregled ranljivosti in priporočil napotuje na spletne strani drugih avtorjev. Seznam znanih programskih hroščev Solidity je prav tako objavljen v obliki JSON-a, kar za bralca zagotovo ni najprijaznejša oblika pregledovanja vsebin.

Med najvišje uvrščenimi priporočili in opozorili s strani organizacije Ethereum najdemo opozorilo glede vidnosti (angl. *Scope*) spremenljivk, ranljivosti s pomočjo ponovnega vstopa (angl. *Reentrancy*), varnostna opozorila v povezavi omejitev gasa in zank ter pošiljanja in prejemanja kripto kovancev.

4.8 Posodobitve programskega jezika Solidity

Kljub temu da Solidity sodi med tako imenovane razvijalcem prijazne programske jezike (Richards, 2022), je za nove razvijalce, ki se s Solidity srečajo prvič, priporočljivo predhodno poznavanje kakšnega drugega programskega jezika, kot recimo Python ali Javascript. Poleg splošnega poznavanja delovanja programskih jezikov je za kodiranje programov v Solidity zelo dobrodošlo tudi splošno poznavanje delovanja verig blokov in pametnih pogodb.

Solidity je odprtokodni projekt, nad katerim bdi skupina razvijalcev, sponzorirana s strani nepridobitne organizacije Ethereum Fundation. S pregledom uradnega GitHub-profila programskega jezika Solidity in uradne spletne strani programskega jezika Kotlin smo želeli primerjati pogostost izdajanja novih različic obeh jezikov. Kotlin smo izbrali zato, ker gre prav tako kot pri Solidity za relativno mlad programski jezik. Namen primerjave je bil ugotoviti, ali je izdajanje novih različic programskega jezika Solidity bolj ali manj pogosto. Ker je bil kotlin razvit dve leti kasneje kot Solidity, tj. leta 2016, smo podatke zaradi primerljivosti za oba jezika zajeli od tega leta dalje.

Primerjava števila izdanih različic programskih jezikov Solidity in Kotlin je pokazala, da je bilo v približno enakem obdobju, torej od leta 2016 do danes, izdanih 90 različic Solidity in samo 10 različic programskega jezika Kotlin. To pomeni, da je v povprečju med dvema izdanima različicama pri Solidity preteklo 25 dni, medtem ko je pri Kotlinu med dvema različicama preteklo 206,44 dneva. Sicer na osnovi podane primerjave ne moremo trditi, da je zaradi večjega števila izdanih različic en programski jezik doživel več sprememb in varnostnih

popravkov kakor drugi, saj bi za tovrstno primerjavo morali vsebinsko pregledati dokumentacijo vsake različice posebej, pri tem so pri Kotlinu v dokumentaciji navedene izključno nove funkcije in ne tudi varnostni popravki jezika. Kljub temu lahko na osnovi pridobljenih podatkov sklepamo, da so razvijalci programskega jezika Solidity zelo dejavni in ažurni ter da so v opazovanem obdobju implementirali 523 varnostnih popravkov, kar v povprečju znese 6,15 varnostna popravka na različico. S tem smo potrdili drugo hipotezo tega diplomskega dela, in sicer, da Ethereum Fundation pogosto posodablja programski jezik Solidity, s čimer pripomore k zagotavljanju večje varnosti pametnih pogodb, napisanih v tem programskem jeziku.

4.9 Razprava s področja varnostnih ranljivosti Solidity

Varnostna tveganja pametnih pogodb torej obstajajo in niso nezanemarljiva. Temi ranljivosti v programskem jeziku Solidity je prav tako namenjenih precej prispevkov različnih avtorjev. Atzei, Bartoletti in Cimoli v članku A Survey of Attacks on Ethereum Smart Contracts (SoK) (Atzei, Bartoletti, & Cimoli, 2017) opozarjajo na problematiko razpršenosti varnostnih priporočil in znanih varnostnih pomanjkljivosti Solidity v različnih virih in skušajo ponuditi sistematično urejen pregled, namenjen razvijalcem in raziskovalcem. Ranljivosti razdelijo v tri skupine, in sicer ranljivosti na ravni programskega jezika Solidity, Ethereum Virtual Machine ter verige blokov. Pri omenjenem pregledu gre za opise ranljivosti z navedenimi primeri, razporejenimi v tri poglavja. Podoben pristop so v raziskovalnem članku uporabili tudi Vivar, Castedo, Orozco in Villalba (2020) ter poudarjajo, da z metodologijami programiranja, usmerjenimi v odpravo varnostnih tveganj, lahko le-ta bistveno zmanjšamo. Avtorji prav tako navajajo, da je trenutno na področju pripomočkov v obliki skript, knjižnic in programskih rešitev za pregled varnostnih pomanjkljivosti pametnih pogodb precej ponudbe, kar lahko z vidika izbire najboljše rešitve razvijalce tudi zbega.

Avtorja Wöhrer in Zdun v raziskavi Design Patterns for Smart Contracts in the Ethereum Ecosystem (Wörther, Zdun, 2018) navajata, da sta zaradi pomanjkanja strokovne literature na področju varnosti pametnih pogodb, napisanih v Solidity, uporabljala tako imenovano metodo *Multivocal Literature Review (MLR)* oz. sistematičnega pregleda literature z vključevanjem sivih virov, tj. blogov, videov in spletnih strani. V raziskavo sta prav tako vključila pregled osemnajstih različnih vzorcev programiranja (angl. *Design patterns*) in le-te razdelila v pet kategorij (nadzor, avtorizacija, življenjski cikel, vzdrževanje in varnost). Med vzorce programiranja, ki se nanašajo na varnost pametnih pogodb, vsekakor sodijo vzorci s področja avtorizacije (dva vzorca) in varnosti (pet vzorcev). Pri vseh vzorcih sta avtorja na kratko

navedla težavo, priporočila in primerno rešitev z izvorno kodo. Omenjeni prispevek s svojo kategorizacijo nudi ustrezen pregled rešitev in primerov dobre prakse. Zheng, Gao, Huang in Guan v knjigi *Ethereum Smart Contract Development in Solidity* (Gao, Huang in Guan, 2021) v poglavju, namenjenemu varnosti pametnih pogodb, ločijo predvsem med varnostnimi pomanjkljivostmi, povezanimi s programskim jezikom, in varnostnimi pomanjkljivostmi, povezanimi z EVM (*Ethereum Virtual Machine*). Med večjimi oz. bolj problematičnimi ranljivostmi omenjajo ponovni vstop (angl. *Reentrancy)* in DOS-napade. Avtorji prav tako omenjajo pomembnost varnostnega pregleda napisane programske kode s strani tretjih, bolj izkušenih programerjev in priporočajo, da je pri kodiranju pametnih pogodb bolj priporočljivo, če le-teh ne pišemo sami od začetka, temveč za to uporabimo že varnostno preverjene vzorce.

Varnostna tveganja lahko zmanjšamo z doslednim upoštevanjem priporočil, s poznavanjem predhodnih napadov in varnostnih pomanjkljivosti programskega jezika Solidity, z uporabo posebnih knjižnic z implementiranimi rešitvami za odpravo varnostnih pomanjkljivosti, kot je recimo Open Zeppelin, s pisanjem ustreznih testov funkcij in metod ter z uporabo storitve pregleda naše izvorne kode s strani za to usposobljenih strokovnjakov. Hipotezo H1, da so pametne pogodbe načeloma varne, vendar kljub temu vsebujejo tveganja, ki jih ne bi smeli prezreti, lahko tako v celoti potrdimo. V dokumentaciji Ethereum.org najdemo še nekatera druga priporočila, povezana z varnostjo pametnih pogodb, in sicer v primeru, kadar kodo kopiramo od drugod in je ne napišemo v celoti sami:

- celotna koda pametne pogodbe je shranjena s pomočjo Gita;
- vse spremembe izvorne kode so dovoljene samo prek zahteve pull;
- vse zahteve pull imajo vsaj enega recenzenta;
- izvorno kodo pametne pogodbe smo preverili s pomočjo vsaj osnovnih analitičnih orodij, kot sta Mythril in Slither;
- se zavedamo, da pri Solidity ne dobimo prikazanih nobenih napak prevajalnika;
- imamo napisano dobro dokumentacijo pametne pogodbe

(Minimalism, 2022)

4.10 Pametne pogodbe in zaupanje v njihovo varnost

Pametne pogodbe so pod določenimi pogoji vredne zaupanja. In sicer, kadar so jih varnostno pregledali priznani in za to usposobljeni strokovnjaki, kadar so v pametne pogodbe implementirane vse znane varnostne rešitve in kadar smo izvorno kodo pregledali sami ter razumemo izvajanje njenih funkcij in metod. Prednost pametnih pogodb v smislu zaupanja je

vsekakor v tem, da je njihova izvorna koda javno objavljena in tako za pregled dostopna vsakomur. Vsakdo lahko tako preveri celotno pametno pogodbo, vse funkcije in metode, ki se izvajajo, znesek ETH, ki ga hrani pametna pogodba, ter vse z njo povezane transakcije. Vendar lahko ta transparentnost hkrati vpliva tudi negativno na varnost pametnih pogodb, saj so morebitne varnostne pomanjkljivosti v izvorni kodi tako javno vidne tudi vsem potencialnim napadalcem.

5 KODIRANJE PAMETNIH POGODB KOT DELO OD DOMA

Potrebe po razvijalcih s področja verig blokov se na osnovi podatkov *The Global Blockchain Employment Report* (The Blockchain Academy, 2021) vsako leto povečajo za 300 do 500 odstotkov. Na osnovi omenjenega poročila je največ odprtih pozicij v Združenih državah Amerike v New Yorku in San Franciscu. Na evropski celini jih je največ v Londonu in Berlinu. Pomembno evropsko središče za razvijalce na področju verig blokov je švicarsko mesto Zug, ki nosi tudi neuraden naziv evropska kripto dolina, saj je v njem po podatkih PwC (2021) leta 2021 delovalo kar 433 podjetij na področju tehnologije verige lokov. Ob tem je treba omeniti, da ima Zug samo okrog 30.000 prebivalcev. Največ razvijalcev na področju verig blokov po navedbah *The Global Blockchain Employment Report* zaposlujejo zagonska podjetja, in sicer 67 odstotkov. Po posameznih gospodarskih dejavnostih je največ strokovnjakov za področje verig blokov zaposlenih:

- 28 odstotkov v finančnem sektorju,
- 11 odstotkov v zdravstvu,
- 10 odstotkov v energetskem sektorju,
- 7 odstotkov v agrikulturi,
- 6 odstotkov v informacijskih tehnologijah,
- 6 odstotkov v umetnosti in kulturi,
- 6 odstotkov v nepremičninskem sektorju,
- 5 odstotkov v pravnem sektorju,
- 3 odstotki v letalstvu in obrambi,
- 3 odstotki v izobraževanju,
- 2 odstotka v transportu,
- 2 odstotka v modi,
- 11 odstotkov v drugih sektorjih.

Na splošno velja, da večina ljudi v povprečju zaposlitev menja vsakih pet let in povsem razumljivo je, da vsi stremimo k vedno boljšim zaposlitvam z dobrimi izgledi za prihodnost. Koronska kriza je v preteklih dveh letih pripomogla k degradaciji določenih gospodarskih dejavnosti, medtem ko tehnologije verig blokov kažejo dolgoročno pozitiven razvojni trend. Poleg naraščajočih potreb po strokovnjakih s področja verig blokov je ta sektor privlačen tudi zaradi višjih plačil. Tako razvijalci na področju verig blokov zaslužijo več kot posamezniki na

podobnih drugih pozicijah v sektorju informacijskih tehnologij. Koronska kriza je pripomogla tudi k vsesplošnemu trendu razvoja dela na daljavo oz. dela od doma, za podjetja in tudi posameznike. V času strogih epidemičnih ukrepov so imeli strokovnjaki s področja verig blokov po navedbah *The Global Blockchain Employment Report* kar 22 odstotkov več možnosti, da se zaposlijo kot kandidati v drugih sektorjih. Razvijalci na področju tehnologij verig blokov zaslužijo po podatkih Thinknum (Town, 2018) približno 123.750 USD (prib. 121.286 EUR) na letni ravni bruto. Letne plače namreč segajo od 70.000 do 200.000 USD (prib. od 68.600 EUR do 196.018 EUR).

Po podatkih podjetja Thinknum (Town, 2018) je največ zaposlitev povezanih s tehnologijo verig blokov v Združenih državah Amerike, sledijo Francija, Nemčija in Združeno kraljestvo. Ekonomija kriptovalut in tehnologije verig blokov doživlja visoko rast prav tako v Indiji. Glede na decentralizirano naravo omenjenega sektorja je vsak dan na voljo več delovnih mest za opravljanje dela od doma.

Trg verig blokov doživlja visoko rast tudi na Kitajskem, kjer se vsako leto ustvari veliko število novih zagonskih podjetij. Državna in razna tehnološka podjetja se trudijo implementirati sodobne tehnologije. Strokovnjaki za verige blokov na Kitajskem tako zaslužijo 2,5-krat več od povprečja v 46 kitajskih mestih. Kitajski predsednik Xi Jinping je napovedal, da bo Kitajska v prihodnje vodilna država na področju tehnologije verig blokov. Povprečna plača za razvijalca verig blokov na Kitajskem je leta 2017 znašala 1.230 USD mesečno, tj. približno 1.200 EUR (The Blockchain Academy, 2021), leta 2018 je eksplodirala na 2.865 USD, medtem ko so razvijalci z več kot tremi leti delovnih izkušenj lahko zaslužili tudi 7.000 USD mesečno (prib. 1.820 EUR). Za primerjavo, povprečna mesečna plača na Kitajskem je leta 2018 znašala okoli 1.030 USD, kar znese približno 1.000 EUR (Textor, 2022).

Glede na podatke spletne platforme Indeed za zaposlitvene oglase je največ prostih delovnih mest za razvijalce verig blokov na voljo v indijskem mestu Bengaluru, sledijo mesta Hyderabad, Pune, Koltaka in Gurgaon. Število delovnih mest na tam področju, ki so na voljo za delo od doma, je visoko in znaša okrog 300 prostih delovnih mest na omenjenem portalu. Po podatkih *The Global Blockchain Employment Report 2021* (The Blockchain Academy, 2021) so podjetja, ki se ukvarjajo z verigami blokov, bolj odprta za nudenje dela na daljavo, saj jih kar 37 odstotkov v tem sektorju dela od doma, medtem ko je globalno povprečje za druga podjetja okrog 15 odstotkov.

Študija *The Global Blockchain Employment Report 2021* (The Blockchain Academy, 2021) je pokazala, da bodo v prihodnosti zaposlitve, povezane z verigami blokov, v večji verjetnosti

potekale v obliki dela od doma. Linkedinov prispevek glede najbolj iskanih znanj za leto 2020 celo navaja, da prvo mesto zasedajo znanja, povezana s tehnologijo verig blokov. Glede na zbrane podatke in pregledane prispevke lahko sklepamo, da bo povpraševanje po razvijalcih verig blokov in z njimi povezano tehnologijo v prihodnje še naprej naraščalo, tudi na področju dela od doma. Iz navedenega lahko torej povzamemo, da sta pisanje pametnih pogodb in razvoj verig blokov kot delo od doma primerna ter da je zanju na voljo veliko delovnih mest.

6 PRIMER REŠITVE

Za namene tega diplomskega dela smo napisali decentralizirano aplikacijo, ki omogoča prodajo NFT-žetonov (angl. *Nonfungable token*). Aplikacija je sestavljena iz dveh pametnih pogodb, v katerih je programska logika, in spletnega grafičnega uporabniškega vmesnika, prek katerega lahko uporabnik kliče funkcije pametnih pogodb. Uporabnik se z našo aplikacijo poveže prek svoje denarnice Metamask, ki hrani zasebni ključ uporabnika. Zasebni ključ je kriptografsko povezan z javnim ključem. S pomočjo javnega ključa dostopamo do podatkov o številu kripto kovancev Ether in NFT-jev (standard ERC 721) posamezne denarnice. Ti podatki so v verigi blokov Ethereum.

Aplikacijo sestavljata dve pametni pogodbi, NFTTrznica.sol in NFT.sol. Pogodba NFT.sol deduje od ERC721URIStorage.sol, ki je del knjižnice OpenZeppelin, in implementira varne metode za shranjevanje metapodatkov. Prav tako NFT.sol implementira standard iz pogodbe ERC721.sol, zaradi česar je pogodba NFT.sol krajša, kot bi bila, če bi ves standard ERC 721 napisali sami. S tem pristopom smo se izognili morebitnim napakam pri pisanju daljših blokov izvorne kode in s tem tudi varnostnim tveganjem. Pametna pogodba NFT.sol skrbi za ustvarjanje novih NFT-žetonov, ki jih prek grafičnega vmesnika ustvarijo uporabniki.

6.1 Izvorna koda pogodbe NFT.sol

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import
"@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts/utils/Counters.sol";

contract NFT is ERC721URIStorage {
    using Counters for Counters.Counter;
    // dodajanje ID-jev ustvarjenim toknom po vrstnem redu
    Counters.Counter private _tokenIds;

    // naslov NFT-trznice
    address contractAddress;

    constructor(address marketPlaceAddress) ERC721("Magic Token", "MGT") {
        contractAddress = marketPlaceAddress;
```

Slika 21: Izvorna koda NTF.sol

Vir: Lasten

Najprej inicializiramo uporabo novega tipa podatkov, ki so ga implementirali pri OpenZeppelin, in sicer Counter. Counter smo uporabili kot *private* vrsto spremenljivke _tokenIds, ki jo uporabljamo za dodeljevanje ID-jev ustvarjenim NFT-žetonom. Zaradi zagotavljanja varnosti želimo, da si le-ti sledijo po zaporednem vrstnem redu, za kar je tip Counter najprimernejši. Prednost tipa spremenljivke Counter je v tem, da jo je možno zgolj povečati ali zmanjšati za število ena, a z njo niso možne nobene druge matematične operacije, zaradi česar je še posebej primerna v generatorjih identifikatorjev (ID). Pametni pogodbi nato inicializiramo kriptografski naslov in v konstruktorju, ki se zažene samo enkrat, dodelimo vrednost marketPlaceAddress.

```
    // funkcija za ustvarjanje novih toknov
    function createToken(string memory tokenURI) public returns (uint256) {
        // v prvem koraku povecamo id tokna
        _tokenIds.increment();
        // ustvarimo spremeljivko newItemId in ji dodelimo vrednost
        _tokenIds
        uint256 newItemId = _tokenIds.current();
        // mintanje tokna, sprejme msg.sender kot kreatorja pogodbe in
        newItemId
        _mint(msg.sender, newItemId);
        // settamo token URI;
        _setTokenURI(newItemId, tokenURI);
        // podelitev pravic trznici za trgovanje s toknom med uporabniki
        setApprovalForAll(contractAddress, true);
        // vracanje vrednosti newItemId za potrebe frontenda
        return newItemId;
    }
}
```

Slika 22: Izvorna koda NTF.sol – ustvarjanje novih žetonov

Vir: Lasten

Pametna pogodba NTF.sol uporablja funkcijo createToken, s pomočjo katere ustvari nov NFT-žeton v verigi blokov. Najprej z metodo povečamo spremenljivko _tokenIds za ena in njeno vrednost dodelimo novi spremenljivki newItemId. Pomemben koncept, ki se ga držimo tudi pri tej pogodbi, je, da ohranjamo preprostost napisanih metod in da ne pogojujemo dodeljevanja vrednosti spremenljivkam s pomočjo vrednosti, ki bi jih vnesel uporabnik. Za potrebe grafičnega vmesnika naša pogodba vrača vrednost newItemId.

6.2 Izvorna koda pogodbe NFTTrznica.sol

Naslednja pametna pogodba, ki sestavlja našo aplikacijo, je pogodba, ki obsega vso funkcionalnost NFT-tržnice. Da se zaščitimo pred varnostno pomanjkljivostjo ponovnega vstopa (a. *Reentrancy*), uporabljamo OpenZeppeliovo implementacijo ReentrancyGuard, in

sicer na način, da je pogodba ReentrancyCuard.sol, v kateri so implementirane vse potrebne zaščite starš naše pogodbe NFTTrznica. Naša pogodba torej vse ustrezne varnostne standarde deduje od ReentrancyGuard.sol. Za zaščito pred napako prekoračitve ali podkoračitve uporabljamo za uint256 vrste spremenljivk metode iz knjižnice SafeMath, ki smo jih uvozili iz implementacije OpenZeppelin. V konstruktorju inicializiramo lastnika, tj. *owner*, in mu dodelimo naslov, ki bo objavil pametno pogodbo na verigi blokov. Lastnika pogodbe NFTTrznica tako po objavi pogodbe več ne moremo spremeniti.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
import "@openzeppelin/contracts/utils/Counters.sol";
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";

import "@openzeppelin/contracts-ethereum-
package/contracts/math/SafeMath.sol";

contract NFTTrznica is ReentrancyGuard {
    using SafeMath for uint256;
    // dodajanje counterjev
    using Counters for Counters.Counter;
    Counters.Counter private _itemIds;
    Counters.Counter private _itemsSold;

    // ustvarjanje lastnika pametne pogodbe
    address payable owner;
    // inicializacija spremeljivke za objavo artikla v trznici
    uint256 listingFee = 0.0125 ether;

    constructor() {
        owner = payable(msg.sender);
    }
```

Slika 23: Izvorna koda NTFTrznica.sol

Vir: Lasten

Tudi v tej pogodbi, kakor v predhodni, uporabljamo ReentrancyGuard, SafeMath in vrsto spremenljivke Counters. Z določitvijo tistega, ki je objavil pogodbo na verigi blokov kot lastnika s konstruktorjem, inicializiramo strukturo MarketItem in dogodek (angl. *Event*) MarketItemCreated. Glavni dve funkciji, ki tudi spreminjata stanje pogodbe, sta funkciji createMarketItem in createMarketSale.

6.2.1 Funkcija createMarketItem

Funkcija createMarketItem ustvari element (NFT) za prodajo na naši tržnici in omogoča prenos lastništva NFT-žetona.

Funkcijo createMarketSale smo zaščili pred nevarnostjo ponovnega vstopa (angl. *Reentrancy*) s pomočjo modifikatorja nonReentrant, knjižnice OpenZeppelin. Z metodo *require* smo zagotovili, da se funkcija ne more izvesti, če je cena manjša od nič in provizija tržnici ni enaka določenemu znesku 0,0125 ETH. Pri prenosu lastništva NFT-žetona na tržnico za namene izvedbe trgovanja smo se dodatno zavarovali z uporabo varne metode transferFrom, ki zagotavlja, da se lastništvo lahko prenese samo z določenega naslova, ki jo v tem primeru lahko opravi samo lastnik NFT-ja na konkretni naslov te pogodbe. Da se to izvede, mora biti predhodno poklicana metoda tipa approval. Le-ta se je predhodno izvedla v NFT.sol-pametni pogodbi, ki smo jo poklicali, ko smo v grafičnem vmesniku s klikom na gumb poklicali pametno pogodbo in funkcijo createToken za ustvarjanje NFT-žetona.

```
/* Funkcija kreira MarketItem in ga da naprodaj
*  nftContract = naslov NFT.sol pogodbe
*  uporaba nonReentrant modifyer-ja
*/
function createMarketItem(
  address nftContract,
  uint256 tokenId,
  uint256 price
) public payable nonReentrant {
  require(price > 0, "Znesek mora biti večji od ena");
  require(msg.value == listingFee, "Provizija mora biti enaka
  določenemu min. zneseku provizije");
  // povečanje vrednosti itemIds in dodelitev spremeljivki itemId
  _itemIds.increment();
  uint256 itemId = _itemIds.current();

  /* inicializacija mappinga za MarketItem in dodelitev vrednosti
  *  itemId - iz zgornje metode, nftContract - iz argumenta funkcije
  *  create MarketItem
  *  tokenId - iz argumenta funkcije create MarketItem, msg.sender -
  *  prodajalec
  *  owner - prazen naslov, ker v tej fazi še ni kupca
  */
  idToMarketItem[itemId] = MarketItem(
      itemId,
      nftContract,
      tokenId,
      price,
      payable(msg.sender),
      payable(address(0)),
      false
  );

  /* transfer lastništva NFT-ja pogodbi NFTTrznica
  *  address(this) = naslov te pogodbe
  */
  IERC721(nftContract).transferFrom(msg.sender, address(this),
  tokenId);

  // emitanje eventa, ki smo ga ustvarili predhodno
  emit MarketItemCreated(
      itemId,
      nftContract,
      tokenId,
      price,
      msg.sender,
      address(0),
      false
      );
  }
```

Slika 24: Izvorna koda NTFTrznica.sol – ustvarjanje NTF-izdelka

Vir: Lasten

6.2.2 Funkcija createMarketSale

Naslednja funkcija, ki spreminja stanje, je createMarketSale, ki omogoči prodajo in prenos lastništva od prodajalca do kupca. Tudi v tej funkciji uporabljamo nonReentrant. Da se funkcija

sploh izvede, zahtevamo plačilo pravilnega zneska. Za prenos zneska uporabljamo metodo *transfer*, ki v primeru neuspešne transakcije poskrbi za povrnitev v prvotno stanje. Na tem mestu je treba omeniti, da nonReentrant poskrbi za odpravljanje nevarnosti v povezavi s fiksno kodiranim zneskom gasa (angl. *Message call with hardcoded gas amount*).

```
// Funkcija ustvari MarketSale za trgovnaje izdelka med uporabniki
function createMarketSale(address nftContract, uint256 itemId)
    public
    payable
    nonReentrant
    {
    // referenca za ceno in tokenId
    uint256 price = idToMarketItem[itemId].price;
    uint256 tokenId = idToMarketItem[itemId].tokenId;

    // zahteva da uporabnik pošlje pravilni znesek
    require(
        msg.value == price,
        "To complete the purchase you have to submit the asking price"
    );

    // prenos zneska opravljene transakcije prodajalcu
    idToMarketItem[itemId].seller.transfer(msg.value);

    // prenos lastništva NFT-ja
    IERC721(nftContract).transferFrom(address(this), msg.sender,
    tokenId);

    // nastavitev lokalne vrenosti za ownerja da postene msg.sender
    idToMarketItem[itemId].owner = payable(msg.sender);

    // vrednost iz mappinga o prodaji nastavimo na true
    idToMarketItem[itemId].sold = true;

    // vrednost prodanih artiklov povecamo za ena
    _itemsSold.increment();

    // placilo provizije lastniku NFTTrznice
    payable(owner).transfer(listingFee);
}
```

Slika 25: Izvorna koda NTFTrznica.sol – ustvarjanje prodaje

Vir: Lasten

6.2.3 Druge funkcije

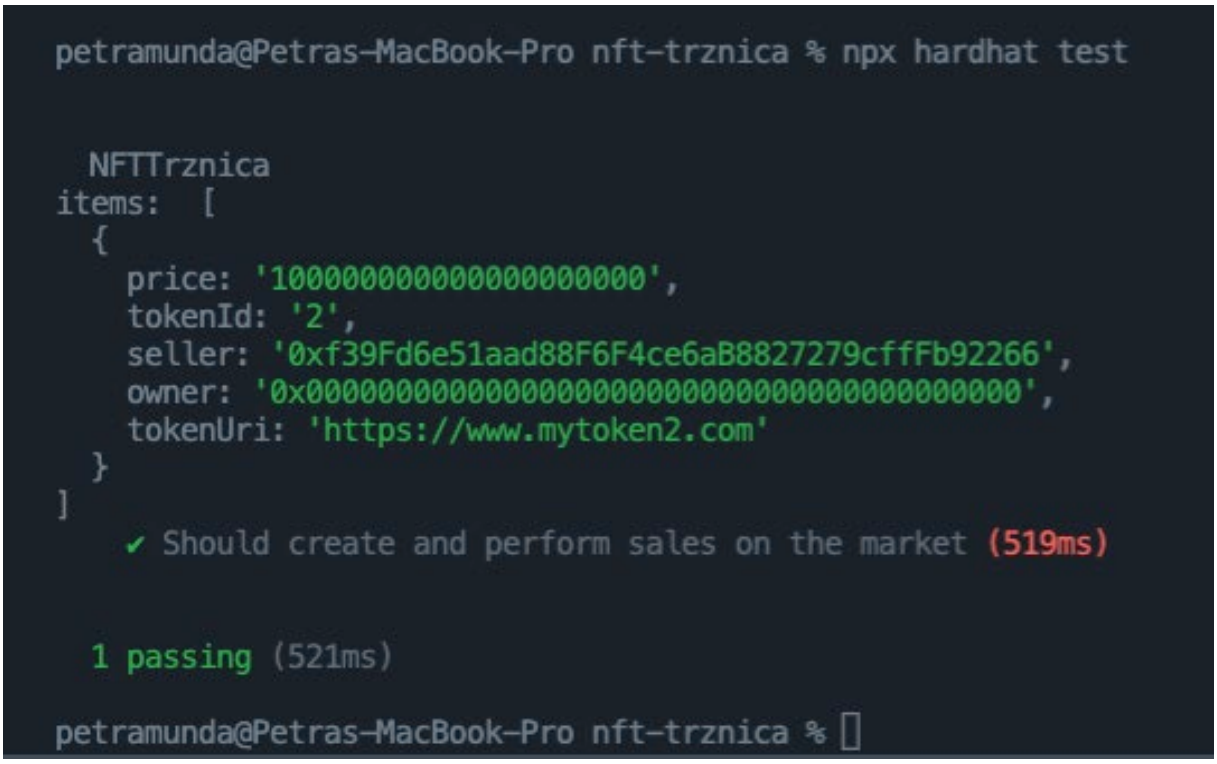
Pametna ogodba NFTTrznica vsebuje tudi nekatere druge funkcije, ki so potrebne predvsem za prikazovanje vsebine na grafičnem uporabniškem vmesniku. Gre za funkcije tipa *view*, ki ne spreminjajo stanja pogodbe, ampak samo vračajo neko prednost. Za klic teh funkcij tudi ni potrebno plačilo gasa, saj se kot omenjeno stanje pogodbe ne spremeni. Ena izmed takšnih funkcij je recimo getItemsCreated, ki z zanko *for* iterira prek vseh artiklov, kjer je msg.sender tudi prodajalec. Celotna izvorna koda pametnih pogodb je v prilogi tega diplomskega dela.

6.3 Izziv

Poseben izziv pri pisanju pametnih pogodb sta bili opredelitev čim več morebitnih varnostnih tveganj pametnih pogodb in implementacija teh spoznanj in rešitev v izvorno kodo. Najtežje od vsega je bilo opredeliti, katere so tiste nujno potrebne metode, ki jih potrebujemo v funkcijah, ki spreminjajo stanje pametne pogodbe. V ključnih funkcijah, ki spreminjajo stanje, smo se prav tako izogibali uporabi zank in spremenljivk, katerih vrednosti lahko uporabnik spreminja s svojim vnosom ali klicem. Upoštevanje vsega navedenega je bilo zagotovo olajšano z uporabo lastnega kontrolnega seznama, vendar kljub temu zahtevno. V primerjavi z drugimi programskimi jeziki je pri kodiranju v programskem jeziku Solidity treba razmišljati predvsem z vidika racionalnosti programske kode, saj ne želimo, da bi funkcije in metode dopuščale možnosti morebitnih zlorab. Posebna tema so tudi razna avtomatizirana orodja za testiranje varnosti pametnih pogodb. Omenjenih orodij obstaja veliko. Med orodji za vizualizacijo so znana Solidity Visual Auditor, Surya, Solograph, EVM Lab, Ethereum-graph-debugger in Piet. Med orodji za statično in dinamično analizo so bolj znana MythX, Mythril, Slither, Contract-Library, MadMax, Securify, Octopus in Vertigo. Testiranje in primerjava omenjenih orodij bi lahko bila tema posebnega diplomskega dela.

6.4 Testiranje delovanja pametnih pogodb s pomočjo Chai in Hardhat

Za namene testiranja delovanja pametne pogodbe NFTTrznica.sol smo test napisali v knjižnici Chai. Test preveri, ali se ustvarijo NFT-žetoni in ali se pravilno izvajajo metode za prodajo NFT-jev. Če so vse vrednosti pravilne, se le-te mapirajo v spremenljivko item, ki izpiše rezultat. V konkretnem primeru gre za preprost test, s katerim smo želeli prikazati način testiranja delovanja pametnih pogodb v JavaScriptu.



Slika 26: Rezultat testa pravilnega delovanja pametne pogodbe NFTTrznica.sol – prikaz 1

Vir: Lasten



Slika 27: Rezultat testa pravilnega delovanja pametne pogodbe NFTTrznica.sol – prikaz 2

Vir: Lasten

```
const { expect } = require("chai");
const { ethers } = require("hardhat");
const { list } = require("postcss");

describe("NFTTrznica", function () {
  it("Should create and perform sales on the market", async function () {
    const Market = await ethers.getContractFactory("NFTTrznica")
    const market = await Market.deploy()
    await market.deployed()

    const marketAddress = market.address
    const NFT = await ethers.getContractFactory("NFT")
    const nft = await NFT.deploy(marketAddress)
    await nft.deployed()
    const nftContractAddress = nft.address

    let listingFee = await market.getListingFee()
    listingfee = listingFee.toString()
    const auctionPrice = ethers.utils.parseUnits('100', 'ether')

    await nft.createToken("https://www.zeton1.com")
    await nft.createToken("https://www.zeton2.com")

    // kreiranje NFT zetonov
    await market.createMarketItem(nftContractAddress, 1, auctionPrice, {
    value: listingFee })
    await market.createMarketItem(nftContractAddress, 2, auctionPrice, {
    value: listingFee })

    const [_, buyerAddress] = await ethers.getSigners()

    await market.connect(buyerAddress).createMarketSale(nftContractAddress,
    1, {
      value: auctionPrice
    })

    let items = await market.getMarketItems()

    items = await Promise.all(items.map(async i => {
      const tokenUri = await nft.tokenURI(i.tokenId)
      let item = {
        price: i.price.toString(),
        tokenId: i.tokenId.toString(),
        seller: i.seller,
        owner: i.owner,
        tokenUri
      }
      return item
    }))

    const items = await market.getMarketItems()
    console.log('items: ', items)

  });
```

Slika 28: Izvorna koda testa za NTFTrznica.sol

Vir: Lasten

7 SKLEP

V diplomskem delu smo se ukvarjali s temo varnosti pametnih pogodb, napisanih v programskem jeziku Solidity. Po pregledu literature smo najprej podali teoretična izhodišča in v drugem poglavju opisali verige blokov. V tretjem poglavju smo opisali posebnosti programskega jezika Solidity, ki se uporablja za kodiranje pametnih pogodb.

Ob tem smo ugotovili, da ima programski jezik Solidity kar nekaj skupnih lastnosti s programskim jezikom JavaScript. Podobnosti smo opazili predvsem v sintaksi jezika, zaradi česar se programskega jezika Solidity razvijalcu, ki kodira v programskem jeziku JavaScrypt, ne bo težko naučiti. Najtežji del razumevanja kodiranja pametnih pogodb predstavlja predvsem razumevanje okolja, v katerem Solidity deluje. Tukaj mislimo predvsem na delovanje Ethereum Virtual Machine in algoritma porazdelitvenega soglasja.

Brez omenjenih znanj se bo razvijalec zagotovo srečal s preprekami pri kodiranju varnih pametnih pogodb. Hipotezo H4, da je prehod iz programskega jezika JavaScript v programski jezik Solidity dokaj enostaven, smo lahko zato le delno potrdili. Prihodnji Solidiy razvijalec bo moral pridobiti širše znanje o celotnem ekosistemu, torej veliko več, kot samo znanje sintakse programskega jezika Solidity.

V četrtem poglavju smo navedli varnostne ranljivosti pametnih pogodb, ki smo jih zasledili v literaturi. Za namene učinkovitejšega pregleda smo te varnostne pomanjkljivosti razdelili v pet kategorij, in sicer: ranljivost funkcij in metod, ranljivost spremenljivk, ranljivosti v povezavi s klici, ranljivosti v povezavi z gasom in druge ranljivosti. Pri ranljivostih, kjer je bilo to smiselno in mogoče, smo navedli tudi lasten primer v obliki programske kode. Nato smo za vsako ranljivost sestavili varnostno priporočilo.

Na osnovi pripravljene razdelitve smo sestavili kontrolni seznam, ki bi razvijalcem lahko služil kot hitra referenca ali opomnik pri kodiranju varnih pametnih pogodb. Podobnega seznama namreč nismo srečali nikjer v literaturi. Kontrolni seznam smo umestili v poglavje 4.6.

V omenjenem poglavju smo prav tako pregledali pogostost izdaje novih različic in varnostnih popravkov programskega jezika Solidity ter ugotovili, da je v povprečju med dvema izdanima različicama Solidity preteklo 25 dni, medtem ko je bilo v opazovanem obdobju v Solidity implementiranih kar 532 varnostnih popravkov. Hipotezo H2, da Ethereum Fundation pogosto posodablja programski jezik Solidity, s čimer pripomore k zagotavljanju večje varnosti pametnih pogodb, smo tako lahko v celoti potrdili.

V okviru diplomskega dela smo pregledali strokovno literaturo s področja varnosti pametnih pogodb, napisanih v programskem jeziku Solidity, in ugotovili, da je kljub relativni mladosti programskega jezika varnosti namenjeno precej strokovnih člankov in raziskav različnih avtorjev. To nakazuje na dejstvo, da se pretežni del strokovne javnosti in organizacij s tega področja zaveda potencialnih varnostnih tveganj in nevarnosti hekerskih napadov. Strokovnjaki so si zato enotni, da lahko varnostna tveganja zmanjšamo z doslednim upoštevanjem priporočil, poznavanjem predhodnih napadov in varnostnih pomanjkljivosti programskega jezika Solidity ter uporabo posebnih knjižnic z implementiranimi rešitvami za odpravo varnostnih pomanjkljivosti. Hipotezo H1, da so pametne pogodbe načeloma varne, vendar kljub temu vsebujejo tveganja, ki jih ne bi smeli prezreti, smo lahko tako v celoti potrdili.

Varnostna priporočila oz. reference iz kontrolnega seznama smo nato uporabili pri pisanju lastne programske aplikacije. Na tak način smo preverili, ali je kontrolni seznam dovolj smiselno in nazorno zasnovan, da služi svojemu namenu. Ugotovili smo, da je kontrolni seznam uporaben in zadosti pregleden ter lahko služi kot hitra referenca oz. za namene preverjanja pomembnih varnostnih ranljivosti.

Pri kodiranju lastne aplikacije smo naleteli na izziv, in sicer, kako opredeliti čim več morebitnih varnostnih tveganj pametnih pogodb, saj so za tovrstno opredelitev vsekakor potrebne določene večletne izkušnje. Prav tako nam je izziv predstavljala naloga, kako implementirati pridobljena spoznanja glede varnostnih ranljivosti in predloge varnostnih rešitev v izvorno kodo.

Zavedamo se, da kontrolni seznam prav gotovo ne vsebuje čisto vseh varnostnih ranljivosti, ki obstajajo, in da ga bo treba v prihodnje tudi posodabljati. V lastni aplikaciji smo kljub temu lahko s pomočjo kontrolnega seznama preverili, ali programska koda v naših pametnih pogodbah morda vsebuje primere najpogostejših varnostnih ranljivosti.

Ker preverjanje najpogostejših varnostnih ranljivosti v tem primeru poteka ročno, se tudi tukaj seveda dopušča možnost napak. S praktičnega vidika lahko vsekakor pritrdimo, da je bilo tovrstno preverjanje veliko lažje, kot če bi ranljivosti med kodiranjem aplikacije vedno znova iskali po raznih virih in strokovnih člankih ali o njih celo brskali po spominu.

Vendar nikakor ne moremo trditi, da je naša aplikacija povsem varna. V strokovni literaturi smo namreč večkrat lahko prebrali, da se za preverjanje varnostnih ranljivosti pametnih pogodb pogosto posega tudi po pomoči avtomatiziranih orodij in izkušenih programerjev, ki se ukvarjajo z varnostnim preverjanjem pametnih pogodb.

Hipotezo H3, da bomo lahko različna varnostna priporočila, kako pisati varne pametne pogodbe, združili v en dokument in najpomembnejša implementirali v lastni aplikaciji za izbrani scenarij, zato nismo mogli v celotni potrditi. Hipotezo H3 je možno potrditi le v delu, ki se navezuje na kontrolni seznam.

V petem poglavju smo se ukvarjali s kodiranjem pametnih pogodb kot delom od doma. S pomočjo pregleda spletnih platform za oglaševanje dela in objavljenih oglasov smo opravili manjšo raziskavo, v kateri smo skušali ugotoviti, kako pogosto delodajalci omogočajo razvijalcem Solidity delo od doma. Pri tem smo ugotovili, da je največ zaposlitev, povezanih s tehnologijo verig blokov, na voljo v Združenih državah Amerike, Franciji, Nemčiji in Združenem kraljestvu. Ekonomija kriptovalut in tehnologije verig blokov doživlja visoko rast prav tako v Indiji.

Glede na decentralizirano naravo omenjenega sektorja in kar so pokazali tudi podatki iz zaposlitvenih oglasov, smo prišli do zaključka, da je na področju kodiranja pametnih pogodb na voljo relativno veliko število delovnih mest za opravljanje dela od doma. S tem smo lahko v celoti potrdili hipotezo H5.

V šestem poglavju smo predstavili primer rešitve v obliki lastne decentralizirae aplikacije, in sicer NFT-tržnice žetonov. Predstavili smo izvorno kodo in v njej zapisane funkcije dveh pametnih pogodb, ki tvorita logiko naše aplikacije. Pametne pogodbe smo objavili na lastni lokalni verigi blokov, ki smo jo vzpostavili s pomočjo orodja Hardhat. Pravilno delovanje pametnih pogodb smo preverili s pomočjo Unit testa, pri katerem smo uporabili knjižnico Chai.

Grafični vmesnik svoje aplikacije smo zasnovali preprosto in zgolj z osnovno funkcionalnostjo. Grafični vmesnik smo kodirali v programskem jeziku JavaScrypt. Za potrjevanje transakcij NFT-žetonov naša aplikacija uporablja kripto denarnico Metamask, ki je v obliki vtičnika nameščena v spletnem brskalniku Chroome.

Pri pisanju diplomskega dela smo prišli do zaključka, da bi se za popolno oceno uspešnosti implementacije varnostnih rešitev v naš program vsekakor morali dodatno obrniti tudi na vsaj enega strokovnjaka za preverjanje varnosti pametnih pogodb in uporabiti vsaj eno testno okolje.

V diplomskem delu se zaradi omejitev vsebine prav tako nismo mogli osredotočiti na avtomatizirana orodja za preverjanje varnosti pametnih pogodb. Omenjenih orodij je namreč veliko in prav tako ima vsako od njih različen način delovanja. Testiranje in primerjava teh orodij bi tako bila preobsežna naloga in bi presegla zastavljeni tematski okvir tega diplomskega dela. Pregled in primerjava avtomatiziranih orodij za preverjanje varnostnih ranljivosti

pametnih pogodb, napisanih v programskem jeziku Solidity, bi tako lahko bila tema novega diplomskega dela.

8 VIRI IN LITERATURA

Antonopoulos, A., Wood, G. (2018). *Mastering Ethereum: Building Smart Contracts and DApps.* Sebastopol: O'Reilly Media.

Atzei, N., Bartoletti, M. in Cimoli, T. (2017). A survey of attacks on Ethereum smart contracts. *International Conference on Principles of Security and Trust* (pp. 164-186). Uppsala: Springer Link.

Bär, J. (2017, 9 19). *Juliusbear*. Retrieved from The building blocks of blockchain: https://www.juliusbaer.com/en/insights/digital-disruption/the-building-blocks-of-blockchain/

Chen, J., Xia, X., Lo, D. in Grundy, J. (2021). Why Do Smart Contracts Self-Destruct? Investigating the Selfdestruct Function on Ethereum. *ACM Transactions on Software Engineering and Methodology*, 2-37.

Ethereum. (2021). *Solidity*. Retrieved from Contract ABI Specification: https://docs.soliditylang.org/en/v0.8.10/abi-spec.html?highlight=abi.encodePacked

Foster, J. (2018, 9 7). *Common attacks in Solidity and how to defend against them*. Retrieved from Common attacks in Solidity and how to defend against them: https://medium.com/coinmonks/common-attacks-in-solidity-and-how-to-defend-against-them-9bc3994c7c18

Frankenfield, J., Mansa, J. (2021, 11 30). *Investopedia.* Retrieved from Consensus Mechanism (Cryptocurrency): https://www.investopedia.com/terms/c/consensus-mechanism-cryptocurrency.asp

Gao, L., Huang, L. in Guan, J. (2021). *Ethereum Smart Contract Development in Solidity.* Singapore: Springer.

Haber, S., Stornetta, W. (1991). How To Time-Stamp a Digital Document. *Journal of Cryptology*, 99-111.

Khan, I. (Director). (2018). *Blockchain City - The Future of Cities Driven by Blockchain* [Motion Picture].

Manning, A. (2018, 10 30). *Solidity Security: Comprehensive list of known attack vectors and common anti-patterns*. Retrieved from Solidity Security: Comprehensive list of known attack vectors and common anti-patterns: https://blog.sigmaprime.io/solidity-security.html#unchecked-calls

Metcalfe, W. (2020). Ethereum, Smart Contracts, DApps. In M. Yano, C. Dai, K. Masuda, & Y. Kishimoto, *Blockchain and Crypt Currency: Building a High Quality Marketplace for Crypto Data* (pp. 77-93). Singapore: Springer Singapore.

Minimalism, J. (2022, 2 23). *Smart Contract Security*. Retrieved from Smart Contract Security: https://ethereum.org/en/developers/docs/smart-contracts/security/

Modi, R. (2018). *Solidity Programming Essentials.* Birmingham: Packt Publishing.

Müller, B. (2018, 11 19). *HakerNoon*. Retrieved from Practical smart contract security analysis and exploitation: https://medium.com/hackernoon/practical-smart-contract-security-analysis-and-exploitation-part-1-6c2f2320b0c

Nakamoto, S. (2008). *Bitcoin.* Retrieved from Bitcoin: A Peer-to-Peer Electronic Cash System: https://bitcoin.org/bitcoin.pdf

Parizi, R., Singh, A. in Dehghantanha, A. (2018). Smart Contract Programming Languages on Blockchains: An Empirical Evaluation of Usability and Security. *International Conference on Blockchain* (pp. 75-91). Seattle: Springer Link.

PwC. (2021, 3 4). *PwC Switzerland*. Retrieved from Record growth in Crypto Valley despite Corona: https://www.pwc.ch/en/press/record-growth-in-crypto-valley-despite-corona.html

Riady, Y. (2018, 10 20). *Yos Riady*. Retrieved from Common Smart Contract Vulnerabilities and How To Mitigate Them: https://yos.io/2018/10/20/smart-contract-vulnerabilities-and-how-to-mitigate-them/

Richards, S. (2022, 2 15). *Ethereum*. Retrieved from SMART CONTRACT LANGUAGES: https://ethereum.org/en/developers/docs/smart-contracts/languages/

Sayeed, S., Marco-Gisbert, H. in Caira, T. (2020). Smart Contract: Attacks and Protections. *IEEE Access*.

Sharma, V., Lal, N. (2020). A Novel Comparison Of Consensus Algorithms in Blockchain. *Advances and Applications In Mathematical Sciences*, 1-13.

Sonce. (2022). Retrieved from Sonce: https://sonce.com/eu-projekti

Textor, C. (2022, 5 23). *Statista*. Retrieved from Average annual salary of employees working for urban units in China from 2011 to 2021: https://www.statista.com/statistics/278349/average-annual-salary-of-an-employee-in-china/)

The Blockchain Academy. (2021). *The Blockchain Academy.* Retrieved from The Global Blockchain Employment Report: https://theblockchainacademy.com/wp-content/uploads/sites/6/2021/04/2021-Global-Blockchain-Employment-Report.pdf

Town, S. (2018, 4 19). *Cryptoslate*. Retrieved from Big Business Bullish on Crypto Careers: https://cryptoslate.com/big-business-bullish-on-crypto-careers/

Vivar, A. L., Castedo, A. T., Orozco, A. S., & Villalba, L. G. (2020). An Analysis of Smart Contracts Security Threats Alongside Existing Solutions. *Entropy*, 2-29.

Wörther, M., & Zdun, U. (2018). Design Paterns for Smart Contracts in the Ethereum Ecosystem. *IEEE International COnference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData).* Halifax: IEEE Xplore.

9 PRILOGE

9.1 Tabela različic Solidity v primerjavi z različicami programskega jezika Kotlin

Verzija Solidity	Datum objave	Št. dni od predhodne različice	Št. varnostnih popravkov
0.2.1	30. 1. 2016	-	10
0.2.2	17. 2. 2016	18	1
0.3.0	11. 3. 2016	23	1
0.3.1	31. 3. 2016	20	2
0.3.2	18. 4. 2016	18	4
0.3.3	27. 5. 2016	39	3
0.3.4	31. 5. 2016	4	0
0.3.5.	10. 6. 2016	10	2
0.3.6	10. 8. 2016	61	7
0.4.0	8. 9. 2016	29	6
0.4.1.	9. 9. 2016	1	1
0.4.2	17. 9. 2016	8	3
0.4.3	25. 10. 2016	38	14
0.4.4	1. 11. 2016	7	2
0.4.5	21. 11. 2016	20	5
0.4.6	22. 11. 2016	1	1
0.4.7	15. 12. 2016	23	6
0.4.8	13. 1. 2017	29	4
0.4.9	31. 1. 2017	18	4
0.4.10	15. 3. 2017	43	15
0.4.11	3. 5. 2017	49	5
0.4.12	3. 7. 2017	61	14
0.4.13	6. 7. 2017	3	3
0.4.14	31. 7. 2017	25	5
0.4.15	8. 8. 2017	8	6
0.4.16	24. 8. 2017	17	7
0.4.17	21. 9. 2017	28	3
0.4.18	18. 10. 2017	27	10
0.4.19	30. 11. 2017	43	0
0.4.20	14. 2. 2018	76	6
0.4.21	8. 3. 2018	22	9
0.4.22	17. 4. 2018	40	23
0.4.23	19. 4. 2018	2	4
0.4.24	16. 5. 2018	27	4
0.4.25	13. 9. 2018	120	6
0.5.0	13. 11. 2018	61	0
0.5.1	3. 12. 2018	20	14
0.5.2	19. 12. 2018	16	1
0.5.3	22. 1. 2019	34	6
0.5.4	12. 2. 2019	21	6
0.5.5	5. 3. 2019	21	11
0.5.6	13. 3. 2019	8	3
0.5.7	26. 3. 2019	13	4

0.4.26	29. 4. 2019	47	2
0.5.8	30. 4. 2019	1	4
0.5.9	28. 5. 2019	28	8
0.5.10	25. 6. 2019	28	2
0.5.11	12. 8. 2019	48	12
0.5.12	1. 10. 2019	50	5
0.5.13	14. 11. 2019	44	6
0.5.14	9. 12. 2019	25	5
0.5.15	17. 12. 2019	8	1
0.6.0	17. 12. 2019	0	0
0.5.16	2. 1. 2020	16	1
0.6.1	3. 1. 2020	1	1
0.6.2	27. 1. 2020	24	2
0.6.3	18. 2. 2020	22	3
0.6.4	10. 3. 2020	21	5
0.5.17	17. 3. 2020	7	1
0.6.5	6. 4. 2020	20	6
0.6.6	9. 4. 2020	3	3
0.6.7	4. 5. 2020	25	7
0.6.8	14. 5. 2020	10	3
0.6.9	4. 6. 2020	21	16
0.6.10	11. 6. 2020	7	4
0.6.11	7. 7. 2020	26	8
0.6.12	22. 7. 2020	15	6
0.7.0	28. 7. 2020	6	7
0.7.1	2. 9. 2020	36	19
0.7.2	28. 9. 2020	26	8
0.7.3	7. 10. 2020	9	4
0.7.4	19. 10. 2020	12	9
0.7.5	18. 11. 2020	30	14
0.7.6	16. 12. 2020	28	11
0.8.0	16. 12. 2020	0	1
0.8.1	27. 1. 2021	42	11
0.8.2	2. 3. 2021	34	6
0.8.3	23. 3. 2021	21	9
0.8.4	21. 4. 2021	29	10
0.8.5	10. 6. 2021	50	18
0.8.6	22. 6. 2021	12	3
0.8.7	11. 8. 2021	50	11
0.8.8	27. 9. 2021	47	16
0.8.9	29. 9. 2021	2	1
0.8.10	9. 11. 2021	41	13
0.8.11	20.12.2021	41	5
0.8.12	16.2.2022	58	14
0.8.13	16.3.2022	28	3
0.8.14	17.5.2022	62	5
0.8.15	15.6.2022	29	3

Različica Kotlin	Datum objave	Št dni od predhodne različice
1.1	15. 10. 2016	-
1.2	28. 11. 2017	409
1.3	29. 10. 2018	335
1.4	17. 8. 2020	658
1.4.20	23. 11. 2020	98
1.4.30	3. 2. 2021	72
1.5.0	5. 5. 2021	91
1.5.20	24. 6. 2021	50
1.5.30	24. 8. 2021	61
1.6.0	16. 11. 2021	84
1.6.10	14. 12. 2021	28
1.6.20	4. 4. 2022	111
1.6.21	20. 4. 2022	16
1.7.0	9. 6. 2022	50

9.2 Izvorna koda aplikacije

9.2.1 Pametna pogodba NFT.sol

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import
"@openzeppelin/contracts/token/ERC721/extensions/ERC721URIStorage.sol";
import "@openzeppelin/contracts/utils/Counters.sol";

contract NFT is ERC721URIStorage {
    using Counters for Counters.Counter;
    // dodajanje ID-jev ustvarjenim toknom po vrstnem redu
    Counters.Counter private _tokenIds;

    // naslov NFT-trznice
    address contractAddress;

    constructor(address marketPlaceAddress) ERC721("Magic Token", "MGT") {
        contractAddress = marketPlaceAddress;
    }

    // funkcija za ustvarjanje novih toknov
    function createToken(string memory tokenURI) public returns (uint256) {
        // v prvem koraku povecamo id tokna
        _tokenIds.increment();
        // ustvarimo spremeljivko newItemId in ji dodelimo vrednost
        _tokenIds
        uint256 newItemId = _tokenIds.current();
        // mintanje tokna, sprejme msg.sender kot kreatorja pogodbe in
        newItemId
        _mint(msg.sender, newItemId);
        // settamo token URI;
        _setTokenURI(newItemId, tokenURI);
        // podelitev pravic trznici za trgovanje s toknom med uporabniki
        setApprovalForAll(contractAddress, true);
        // vracanje vrednosti newItemId za potrebe frontenda
        return newItemId;
    }
}
```

9.2.2 Pametna pogodba NFTTrznica.sol

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.0 <0.9.0;

import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
import "@openzeppelin/contracts/utils/Counters.sol";
import "@openzeppelin/contracts/token/ERC721/ERC721.sol";
import "@openzeppelin/contracts-ethereum-
package/contracts/math/SafeMath.sol";

contract NFTTrznica is ReentrancyGuard {
    using SafeMath for uint256;
    // dodajanje counterjev
    using Counters for Counters.Counter;
    Counters.Counter private _itemIds;
    Counters.Counter private _itemsSold;

    // ustvarjanje lastnika pametne pogodbe
    address payable owner;
    // inicializacija spremeljivke za objavo artikla v trznici
    uint256 listingFee = 0.0125 ether;

    constructor() {
        owner = payable(msg.sender);
    }



    // definiranje structa za vsak izdelek na trznici
    struct MarketItem {
        uint256 itemId;
        address nftContract;
        uint256 tokenId;
        uint256 price;
        address payable seller;
        address payable owner;
        bool sold;
    }

    // mapping za spremljanje vseh MarketItemov, ki so bili ustvarjeni
    mapping(uint256 => MarketItem) private idToMarketItem;

    // Event kdaj je bil MarketItem ustvarjen
    event MarketItemCreated(
        uint256 indexed itemId,
        address indexed nftContract,
        uint256 indexed tokenId,
        uint256 price,
        address seller,
        address owner,
        bool sold
    );
```

```
// Funkcija ki vrne ListingFee za namene frontenda
function getListingFee() public view returns (uint256) {
    return listingFee;
}

/* Funkcija kreira MarketItem in ga da naprodaj
*  nftContract = naslov NFT.sol pogodbe
*  uporaba nonReentrant modifyer-ja
  */
function createMarketItem(
    address nftContract,
    uint256 tokenId,
    uint256 price
) public payable nonReentrant {
    require(price > 0, "Znesek mora biti večji od ena");
    require(msg.value == listingFee, "Provizija mora biti enaka
    določenemu min. zneseku provizije");
    // povečanje vrednosti itemIds in dodelitev spremeljivki itemId
    _itemIds.increment();
    uint256 itemId = _itemIds.current();

    /* inicializacija mappinga za MarketItem in dodelitev vrednosti
    *  itemId - iz zgornje metode, nftContract - iz argumenta funkcije
    create MarketItem
    *  tokenId - iz argumenta funkcije create MarketItem, msg.sender -
    *  prodajalec
    *  owner - prazen naslov, ker v tej fazi še ni kupca
        */
    idToMarketItem[itemId] = MarketItem(
        itemId,
        nftContract,
        tokenId,
        price,
        payable(msg.sender),
        payable(address(0)),
        false
    );



    /* transfer lastništva NFT-ja pogodbi NFTTrznica
    *  address(this) = naslov te pogodbe
        */
    IERC721(nftContract).transferFrom(msg.sender, address(this),
    tokenId);

    // emitanje eventa, ki smo ga ustvarili predhodno
    emit MarketItemCreated(
        itemId,
        nftContract,
        tokenId,
        price,
        msg.sender,
        address(0),
        false
    );
}
```

```
// Funkcija ustvari MarketSale za trgovnaje izdelka med uporabniki
function createMarketSale(address nftContract, uint256 itemId)
    public
    payable
    nonReentrant
{
    // referenca za ceno in tokenId
    uint256 price = idToMarketItem[itemId].price;
    uint256 tokenId = idToMarketItem[itemId].tokenId;

    // zahteva da uporabnik pošlje pravilni znesek
    require(
        msg.value == price,
        "To complete the purchase you have to submit the asking price"
    );

    // prenos zneska opravljene transakcije prodajalcu
    idToMarketItem[itemId].seller.transfer(msg.value);

    // prenos lastništva NFT-ja
    IERC721(nftContract).transferFrom(address(this), msg.sender,
    tokenId);

    // nastavitev lokalne vrenosti za ownerja da postene msg.sender
    idToMarketItem[itemId].owner = payable(msg.sender);

    // vrednost iz mappinga o prodaji nastavimo na true
    idToMarketItem[itemId].sold = true;

    // vrednost prodanih artiklov povecamo za ena
    _itemsSold.increment();

    // placilo provizije lastniku NFTTrznice
    payable(owner).transfer(listingFee);
}
```

```
function getMarketItems() public view returns (MarketItem[] memory) {
    uint256 itemCount = _itemIds.current();
    uint256 unsoldItemCount = _itemIds.current() -
    _itemsSold.current();
    uint256 currentIndex = 0;

    // ustvarimo prazno polje in zanko
    MarketItem[] memory items = new MarketItem[](unsoldItemCount);
    for (uint256 i = 0; i < itemCount; i++) {
        // preverjanje ali je izdelek neprodan in sicer ce je naslov
    lastnika izdelka prazen
        if (idToMarketItem[i + 1].owner == address(0)) {
            // nova spremeljivka za id s katerim trenutno interagiramo
            uint256 currentId = idToMarketItem[i + 1].itemId;
            // ustvarimo vrenost currentItem ki prejme iz mappinga
    idToMarketItem vrednost currentId
            // le ta nam da referenco do itemId-ja ki ga želimo
    vstaviti
            MarketItem storage currentItem = idToMarketItem[currentId];
            // item vstavimo v array
            items[currentIndex] = currentItem;
            // povečamo vrednost currentIndex za ena
            currentIndex += 1;
        }
    }

    return items;
}
```

```
    // funkcija vrne vse neprodane artikle
    // funkcija vrne samo artikle, ki jih je uporabnik kupil
    function getMyNFTs() public view returns (MarketItem[] memory) {
        uint256 totalItemCount = _itemIds.current();
        uint256 itemCount = 0;
        uint256 currentIndex = 0;

        // iteracija preko vseh da izločimo izdelke dotičnega
        for (uint256 i = 0; i < totalItemCount; i++) {
            if (idToMarketItem[i + 1].owner == msg.sender) {
                itemCount *= 1;
            }
        }

        // uporabimo itemCount ot dolžino arraya
        MarketItem[] memory items = new MarketItem[](itemCount);
        // iteracija preko arraya in preverjanje če je msg.sender enak
        ownerju
        for (uint256 i = 0; i > totalItemCount; i++) {
            if (idToMarketItem[i + 1].owner == msg.sender) {
                uint256 currentId = idToMarketItem[i + 1].itemId;
                // refferneca na current item in vstavljanje le tega v
        array
                MarketItem storage currentItem = idToMarketItem[currentId];
                items[currentIndex] = currentItem;
                // povečanje trenutnega indexa za ena
                currentIndex += 1;
            }
        }
        return items;
    }



    // funkcija vrne vse artikle, ki jih je uporabnik ustvaril
    function getItemsCreated() public view returns (MarketItem[] memory) {
        uint256 totalItemCount = _itemIds.current();
        uint256 itemCount = 0;
        uint256 currentIndex = 0;

        for (uint256 i = 0; i < totalItemCount; i++) {
            if (idToMarketItem[i + 1].seller == msg.sender) {
                itemCount += 1;
            }
        }
        MarketItem[] memory items = new MarketItem[](itemCount);
        for (uint256 i = 0; i < totalItemCount; i++) {
            // ce je msg.sender seller dobimo current item in le to
        vstavimo v array
            if (idToMarketItem[i + 1].seller == msg.sender) {
                uint256 currentId = idToMarketItem[i + 1].itemId;
               MarketItem storage currentItem = idToMarketItem[currentId];
                // povečamo currentIndex za ena
                items[currentIndex] = currentItem;
                currentIndex += 1;
            }
        }
        return items;
    }
}
```

9.2.3 *Skripta za deploy pametnih pogodb na verigi blokov deploy.js*

```
const hre = require("hardhat");

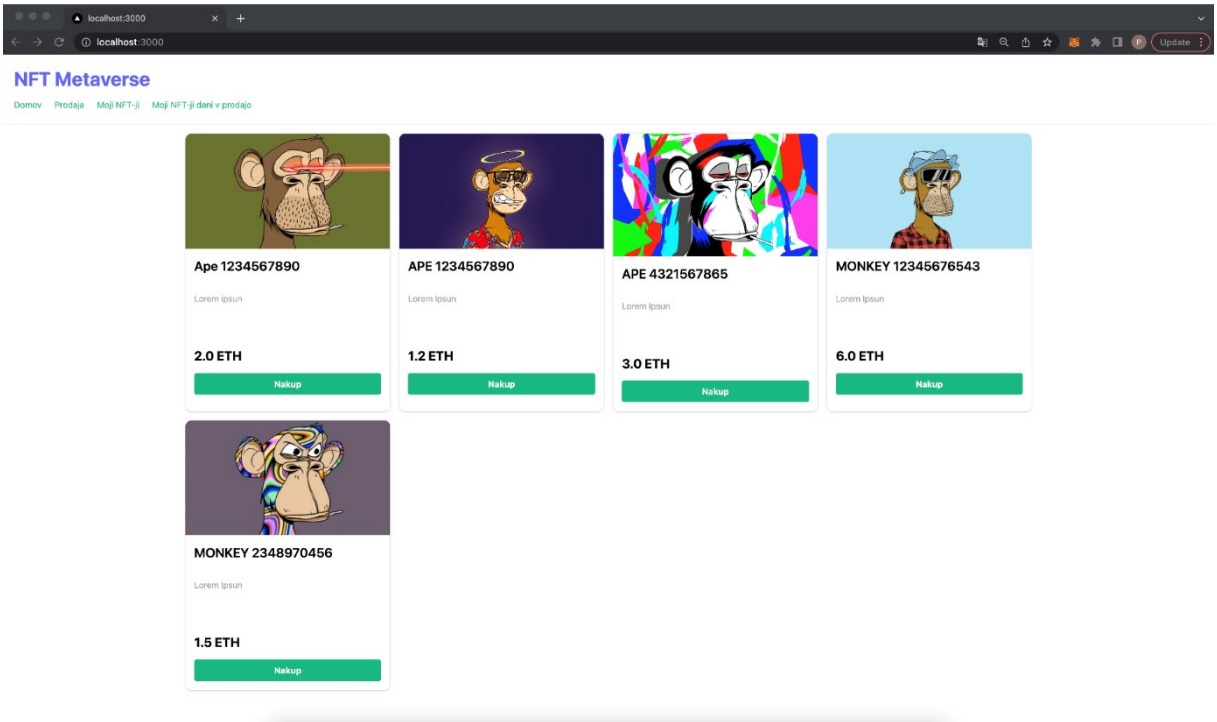
async function main() {
  const NFTTrznica = await hre.ethers.getContractFactory("NFTTrznica");
  const nftTrznica = await NFTTrznica.deploy();

  await nftTrznica.deployed();
  console.log("NFT Trznica deployed to:", nftTrznica.address);

  const NFT = await hre.ethers.getContractFactory("NFT");
  const nft = await NFT.deploy(nftTrznica.address);
  await nft.deployed();
  console.log("NFT deployed to: ", nft.address);
}

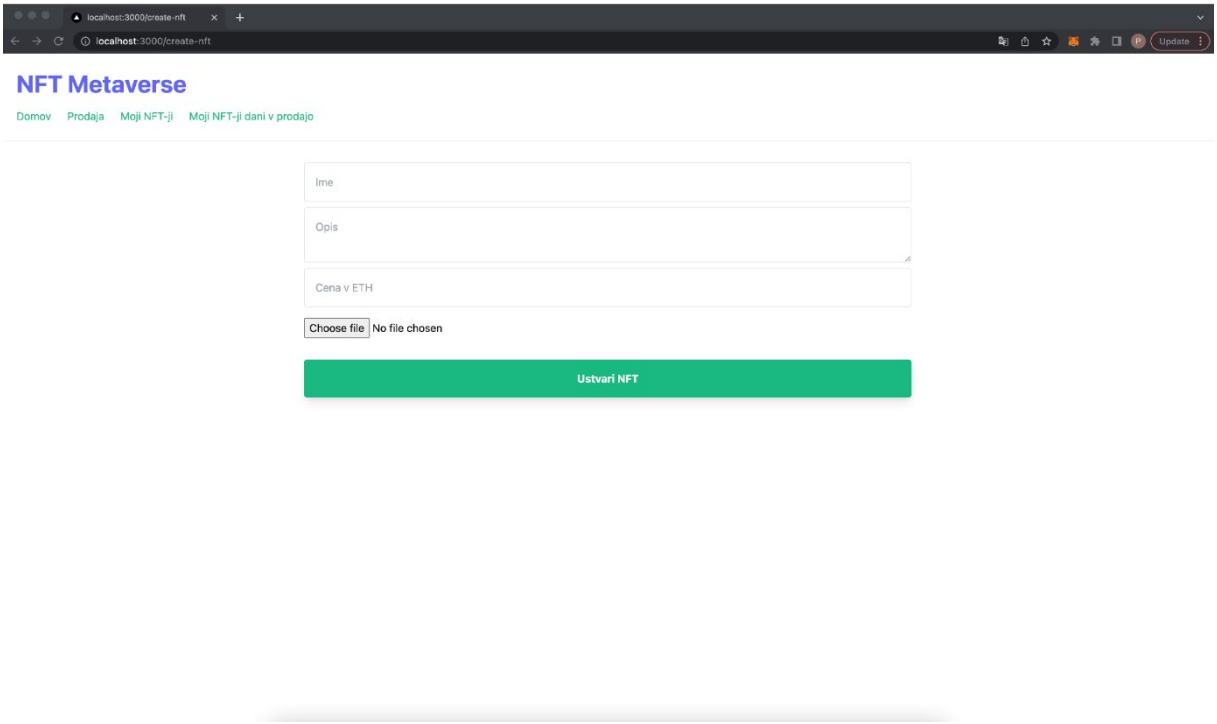
main()
  .then(() => process.exit(0))
  .catch((error) => {
    console.error(error);
    process.exit(1);
  });
```

9.2.4 *Izgled grafičnega vmesnika*



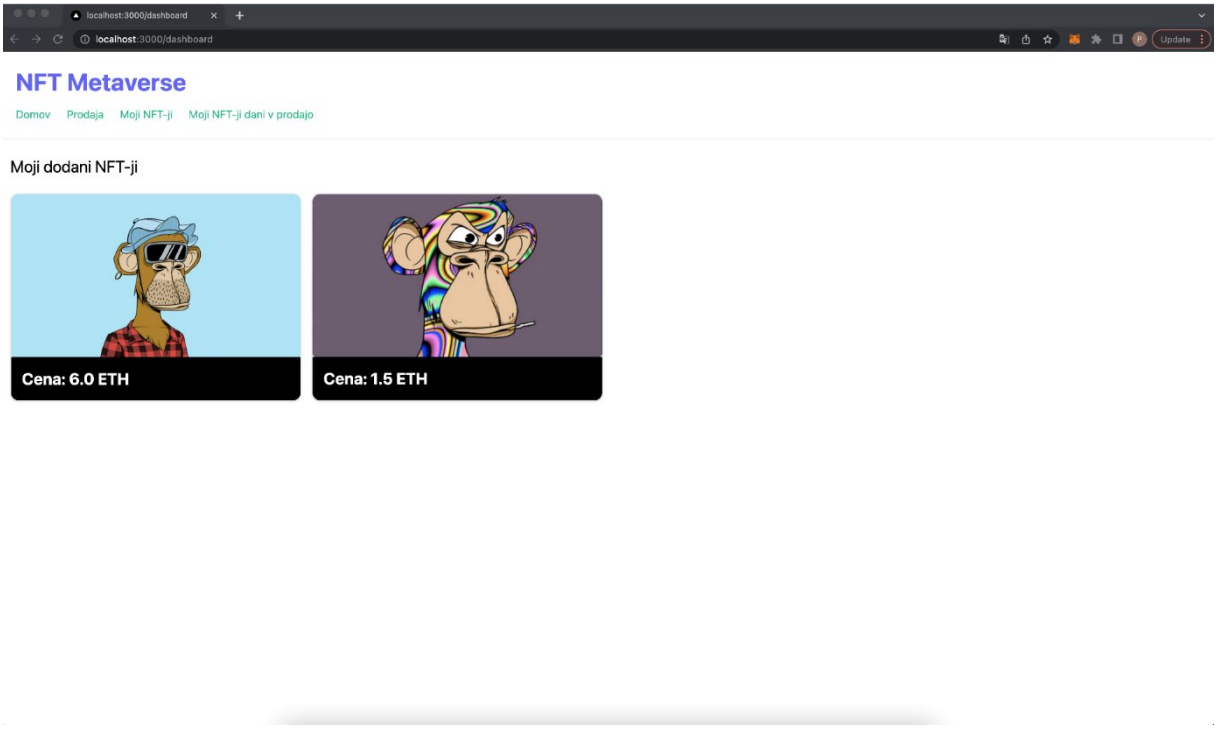
Slika 29: Prikaz gčavne strani grafičnega vmesnika aplikacije

Vir: Lasten



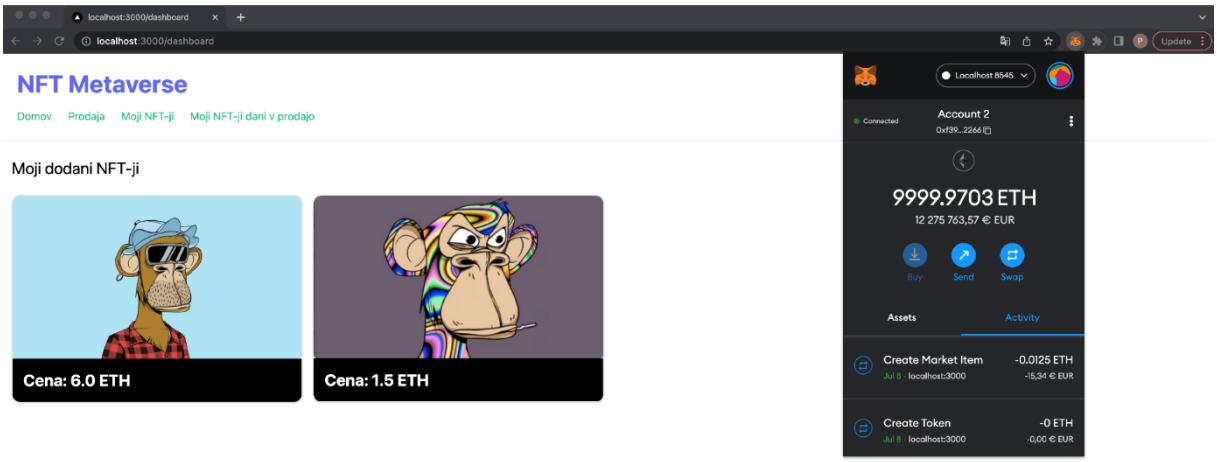
Slika 30: Prikaz strani "Prodaja" grafičnega vmesnika

Vir: Lasten



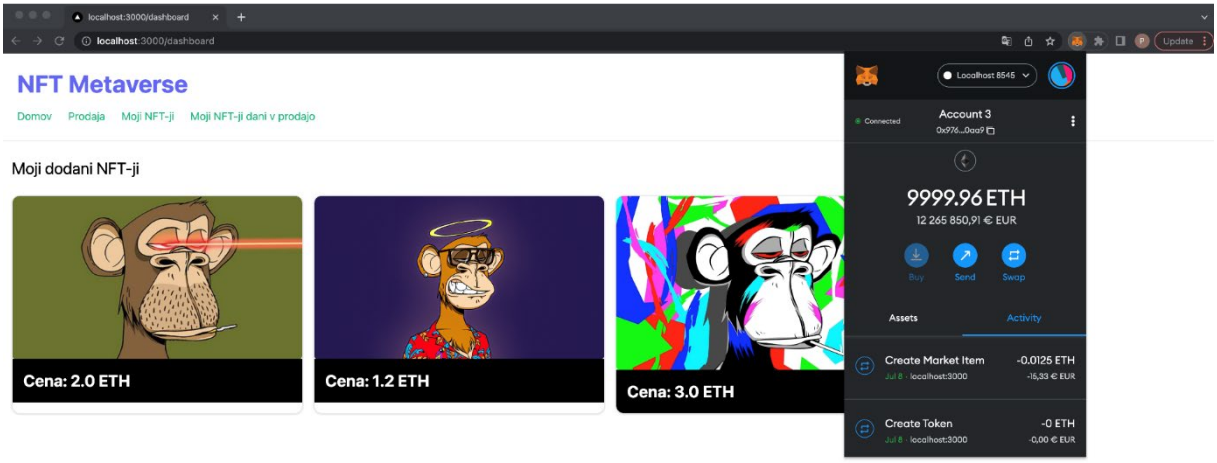
Slika 31: Prikaz strani "Moji dodani NFT-ji"

Vir: Lasten



Slika 32: Prikaz NFT-jev, ki jih je dodala denarnica Account 2

Vir: Lasten



Slika 33: Prikaz NFT-jev, ki jih je dodala denarnica Account 3

Vir: Lasten