

O'REILLY®



Ethereum

dla zaawansowanych

TWORZENIE INTELIGENTNYCH KONTRAKTÓW
I APLIKACJI ZDECENTRALIZOWANYCH



Helion 

Andreas M. Antonopoulos
Dr. Gavin Wood

Tytuł oryginału: Mastering Ethereum: Building Smart Contracts and Dapps

Tłumaczenie: Tomasz Walczak

ISBN: 978-83-283-5574-3

© 2019 Helion S.A.

Authorized Polish translation of the English edition of Mastering Ethereum

ISBN 9781491971949 © 2019 The Ethereum Book LLC, Gavin Wood.

This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

All rights reserved. No part of this book may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying, recording or by any information storage retrieval system, without permission from the Publisher.

Wszelkie prawa zastrzeżone. Nieautoryzowane rozpowszechnianie całości lub fragmentu niniejszej publikacji w jakiegokolwiek postaci jest zabronione. Wykonywanie kopii metodą kserograficzną, fotograficzną, a także kopiowanie książki na nośniku filmowym, magnetycznym lub innym powoduje naruszenie praw autorskich niniejszej publikacji.

Wszystkie znaki występujące w tekście są zastrzeżonymi znakami firmowymi bądź towarowymi ich właścicieli.

Autor oraz Helion SA dołożyli wszelkich starań, by zawarte w tej książce informacje były kompletne i rzetelne. Nie biorą jednak żadnej odpowiedzialności ani za ich wykorzystanie, ani za związane z tym ewentualne naruszenie praw patentowych lub autorskich. Autor oraz Helion SA nie ponoszą również żadnej odpowiedzialności za ewentualne szkody wynikłe z wykorzystania informacji zawartych w książce.

Helion SA

ul. Kościuszki 1c, 44-100 Gliwice

tel. 32 231 22 19, 32 230 98 63

e-mail: helion@helion.pl

WWW: <http://helion.pl> (księgarnia internetowa, katalog książek)

Drogi Czytelniku!

Jeżeli chcesz ocenić tę książkę, zajrzyj pod adres

<http://helion.pl/user/opinie/ethzaa>

Możesz tam wpisać swoje uwagi, spostrzeżenia, recenzję.

Printed in Poland.

- [Kup książkę](#)
- [Poleć książkę](#)
- [Oceń książkę](#)

- [Księgarnia internetowa](#)
- [Lubię to! » Nasza społeczność](#)

Spis treści

Przedmowa	15
Krótki słownik	25
1. Czym jest Ethereum?	37
Porównanie do Bitcoina	37
Komponenty łańcucha bloków	38
Narodziny Ethereum	39
Cztery wersje w rozwoju Ethereum	40
Ethereum — łańcuch bloków o ogólnym przeznaczeniu	41
Komponenty Ethereum	42
Lektura dodatkowa	43
Ethereum i kompletność w sensie Turinga	43
Kompletność w sensie Turinga jako „pożądana cecha”	44
Skutki kompletności w sensie Turinga	44
Od łańcuchów bloków ogólnego użytku do aplikacji DApp	45
Trzecia era internetu	45
Kultura rozwoju Ethereum	46
Dlaczego warto poznać Ethereum?	47
Czego nauczysz się dzięki tej książce?	47
2. Podstawy Ethereum	49
Jednostki waluty ether	49
Wybieranie portfela Ethereum	50
Kontrola i odpowiedzialność	51
Rozpoczynanie pracy z portfelem MetaMask	52
Tworzenie portfela	53
Zmienianie sieci	55

Zdobywanie testowych etherów	56
Wysyłanie etherów z portfela MetaMask	58
Przeglądanie historii transakcji dla adresu	59
Wprowadzenie do światowego komputera	60
Konta EOA i konta kontraktów	61
Prosty kontrakt — kran z testowymi etherami	61
Kompilowanie kontraktu Faucet	64
Tworzenie kontraktu w łańcuchu bloków	66
Interakcja z kontraktem	67
Wyświetlanie adresu kontraktu w eksploratorze bloków	68
Zasilanie kontraktu	68
Wycofywanie środków z kontraktu	69
Podsumowanie	72
3. Klienci Ethereum	73
Sieci Ethereum	74
Czy powinienś uruchamiać pełny węzeł?	74
Wady i zalety pełnych węzłów	75
Wady i zalety publicznych sieci testowych	75
Wady i zalety lokalnego symulowania pracy łańcuchów bloków	76
Uruchamianie klienta Ethereum	77
Wymagania sprzętowe związane z pełnym węzłem	77
Wymagania programowe dotyczące budowania i uruchamiania klientów (węzłów)	78
Klient Parity	79
Klient Go-Ethereum (Geth)	80
Pierwsza synchronizacja łańcuchów bloków opartych na Ethereum	82
Uruchamianie klienta Geth lub Parity	83
Interfejs JSON-RPC	83
Zdalne klienty Ethereum	85
Portfele mobilne (na smartfony)	86
Portfele działające w przeglądarkach	86
Podsumowanie	88
4. Kryptografia	89
Klucze i adresy	89
Kryptografia klucza publicznego a kryptowaluty	90
Klucze prywatne	92
Generowanie klucza prywatnego na podstawie liczby losowej	92

Klucze publiczne	94
Kryptografia krzywej eliptycznej	95
Operacje arytmetyczne na krzywej eliptycznej	97
Generowanie klucza publicznego	98
Biblioteki do pracy z krzywą eliptyczną	99
Kryptograficzne funkcje skrótu	99
Kryptograficzna funkcja skrótu w Ethereum — Keccak-256	101
Z której funkcji skrótu korzystasz?	101
Adresy Ethereum	102
Formaty adresów Ethereum	102
Protokół ICAP	103
Kodowanie szesnastkowe z sumami kontrolnymi opartymi na wielkości liter (EIP-55)	104
Podsumowanie	106
5. Portfele	107
Przegląd technologii używanych w portfelach	107
Portfele niedeterministyczne (losowe)	108
Portfele deterministyczne (z ziarnem)	110
Portfele HD (BIP-32 i BIP-44)	110
Ziarna i kody mnemoniczne (BIP-39)	110
Zalecane praktyki dotyczące portfeli	112
Kody mnemoniczne (BIP-39)	112
Tworzenie portfela HD na podstawie ziarna	118
Portfele HD (BIP-32) i ścieżki (BIP-43/44)	119
Podsumowanie	123
6. Transakcje	125
Struktura transakcji	125
Wartość nonce w transakcji	126
Śledzenie wartości nonce	127
Luki w wartościach nonce, powtarzające się wartości nonce i zatwierdzanie	129
Współbieżność, źródło transakcji i wartości nonce	130
Paliwo dla transakcji	131
Odbiorca transakcji	132
Wartość i dane transakcji	133
Przekazywanie środków do kont EOA i kontraktów	135
Przekazywanie danych do kont EOA lub kontraktów	135

Specjalne transakcje — tworzenie kontraktu	137
Podpisy cyfrowe	139
Algorytm ECDSA	139
Jak działają podpisy cyfrowe?	140
Sprawdzanie poprawności podpisu	140
Obliczenia w algorytmie ECDSA	141
Podpisywanie transakcji w praktyce	142
Tworzenie i podpisywanie nieprzetworzonych transakcji	143
Tworzenie nieprzetworzonych transakcji zgodnych z EIP-155	143
Przedrostek w podpisie (v) i odzyskiwanie klucza publicznego	144
Oddzielanie podpisywania od przesyłania (podpisywanie w trybie offline)	145
Rozsyłanie transakcji	147
Rejestrowanie danych w łańcuchu bloków	147
Transakcje z wieloma podpisami	148
Podsumowanie	148
7. Inteligentne kontrakty i język Solidity	149
Czym jest inteligentny kontrakt?	149
Cykl życia inteligentnego kontraktu	150
Wprowadzenie do wysokopoziomowych języków w Ethereum	151
Tworzenie inteligentnego kontraktu za pomocą Solidity	153
Wybieranie wersji języka Solidity	153
Pobieranie i instalowanie Solidity	154
Środowisko programistyczne	154
Pisanie prostego programu w języku Solidity	155
Kompilowanie przy użyciu kompilatora Solidity (solc)	155
Interfejs ABI kontraktów w Ethereum	155
Wybieranie kompilatora Solidity i wersji języka	156
Programowanie w języku Solidity	157
Typy danych	157
Wbudowane zmienne globalne i funkcje	159
Definicja kontraktu	161
Funkcje	162
Konstruktor kontraktu i polecenie selfdestruct	163
Dodawanie konstruktora i polecenia selfdestruct do kontraktu Faucet	164
Modyfikatory funkcji	165
Dziedziczenie kontraktów	166

Obsługa błędów (assert, require i revert)	168
Zdarzenia	169
Wywoływanie innych kontraktów (polecenia send, call, callcode i delegatecall)	172
Kwestie związane z paliwem	176
Unikanie tablic o dynamicznie określonej wielkości	177
Unikanie wywołań innych kontraktów	177
Szacowanie kosztów paliwa	177
Podsumowanie	178
8. Inteligentne kontrakty i język Vyper	179
Luki a Vyper	179
Porównanie z Solidity	180
Modyfikatory	180
Dziedziczenie klas	181
Wewnątrzwerszowe stosowanie asemblera	181
Przeciążanie funkcji	182
Rzutowanie typów zmiennych	182
Warunki wstępne i końcowe	183
Dekoratory	184
Kolejność funkcji i zmiennych	184
Kompilacja	185
Ochrona przed błędami przepełnienia na poziomie kompilatora	186
Odczyt i zapis danych	186
Podsumowanie	187
9. Bezpieczeństwo inteligentnych kontraktów	189
Zalecane praktyki z zakresu bezpieczeństwa	189
Zagrożenia z obszaru bezpieczeństwa i antywzorce	190
Wielobieżność	190
Praktyczny przykład — The DAO	194
Przepełnienie i niedopełnienie arytmetyczne	194
Przykłady praktyczne — PoWHC i przepełnienie przy transferze zbiorczym (CVE-2018-10299)	198
Nieoczekiwane ethery	198
Więcej przykładów	202
DELEGATECALL	202
Praktyczny przykład — portfel Parity z wielopodpisem (drugi atak)	206

Domyślne poziomy widoczności	207
Praktyczny przykład — portfel Parity z wielopodpisem (pierwszy atak)	208
Złudzenie losowości	209
Praktyczny przykład — kontrakty z generatorami liczb pseudolosowych	210
Korzystanie z zewnętrznych kontraktów	210
Praktyczny przykład — przynęta i wielobieżność	214
Atak związany z krótkimi adresami i parametrami	215
Niesprawdzanie wartości zwracanych przez funkcję call	217
Przykład praktyczny — kontrakty Etherpot i King of the Ether	218
Sytuacje wyścigu i front running	219
Praktyczne przykłady — ERC20 i Bancor	221
Ataki DoS	221
Praktyczny przykład — GovernMental	223
Manipulowanie znacznikiem czasu bloku	224
Praktyczny przykład — GovernMental	225
Ostrożnie z konstruktorami	225
Praktyczny przykład — Rubixi	226
Niezainicjowane wskaźniki do pamięci trwałej	226
Praktyczne przykłady — przynęty OpenAddressLottery i CryptoRoulette	228
Liczby zmiennoprzecinkowe i precyzja	229
Praktyczny przykład — Ethstick	230
Uwierzytelnianie z użyciem zmiennej tx.origin	230
Kontrakty bibliotek	232
Podsumowanie	233
10. Tokeny	235
W jaki sposób tokeny są używane?	235
Tokeny i wymiennność	237
Ryzyko związane z drugą stroną transakcji	237
Tokeny i nieodłączność (wewnętrzność)	237
Używanie tokenów — narzędzia czy aktywa	238
To kaczka!	239
Tokeny narzędziowe — komu są potrzebne?	239
Tokeny w Ethereum	240
Standard ERC20	241
Tworzenie własnego tokenu ERC20	244
Problemy z tokenami ERC20	254

ERC223 — proponowany standard interfejsu kontraktów tokenów	255
ERC777 — proponowany standard interfejsu kontraktów tokenów	256
ERC721 — standard niewymiennych tokenów	258
Stosowanie standardów związanych z tokenami	260
Czym są standardy dotyczące tokenów? Do czego służą?	260
Czy powinieneś stosować opisane standardy?	260
Bezpieczeństwo dzięki dojrzałości	261
Rozszerzenia standardów dotyczących interfejsów tokenów	261
Tokeny i emisje ICO	262
Podsumowanie	263
11. Wyrocznie	265
Dlaczego potrzebne są wyrocznie?	265
Przypadki użycia wyroczni i przykłady	266
Wzorce projektowe dotyczące wyroczni	267
Uwierzytelnianie danych	270
Wyrocznie obliczeniowe	271
Zdecentralizowane wyrocznie	272
Interfejsy klientów wyroczni w języku Solidity	273
Podsumowanie	277
12. Zdecentralizowane aplikacje (DApp)	279
Czym jest aplikacja DApp?	280
Back-end (inteligentny kontrakt)	281
Front-end (internetowy interfejs użytkownika)	281
Przechowywanie danych	282
Zdecentralizowane protokoły przekazywania komunikatów	282
Prosta przykładowa aplikacja DApp — Auction	283
Aplikacja Auction — inteligentne kontrakty back-endu	284
Aplikacja Auction — front-endowy interfejs użytkownika	287
Dalsze decentralizowanie aplikacji Auction	288
Zapisywanie aplikacji Auction w systemie Swarm	289
Przygotowywanie systemu Swarm	289
Przesyłanie plików do systemu Swarm	290
Usługa Ethereum Name Service (ENS)	292
Historia usługi ENS	292
Specyfikacja usługi ENS	292

Dolna warstwa — właściciele nazw i resolwery	293
Warstwa pośrednia — węzły .eth	295
Najwyższa warstwa — tokeny deed	296
Rejestrowanie nazwy	297
Zarządzanie nazwą w usłudze ENS	300
Resolwery w usłudze ENS	301
Tłumaczenie nazwy na skrót w systemie Swarm (treść)	302
Od tradycyjnych aplikacji do aplikacji DApp	303
Podsumowanie	304
13. Maszyna wirtualna Ethereum	305
Czym jest maszyna EVM?	305
Porównanie z istniejącymi technologiami	307
Zbiór instrukcji maszyny EVM (operacje w kodzie bajtowym)	307
Stan w Ethereum	310
Kompilowanie kodu w języku Solidity do kodu bajtowego maszyny EVM	311
Kod do instalowania kontraktu	314
Dezasemblacja kodu bajtowego	315
Kompletność w sensie Turinga a paliwo	319
Paliwo	320
Obliczanie zużycia paliwa w trakcie wykonywania kodu	320
Uwagi związane z obliczaniem zużycia paliwa	321
Zużycie paliwa a cena paliwa	321
Limit paliwa dla bloku	322
Podsumowanie	323
14. Konsensus	325
Konsensus oparty na dowodach pracy	326
Osiąganie konsensusu na podstawie dowodów stawki	326
Ethash — algorytm dowodów pracy w Ethereum	327
Casper — algorytm dowodów stawki dla Ethereum	328
Reguły osiągania konsensusu	329
Kontrowersje i konkurencja	329
Podsumowanie	330
A Historia forków w Ethereum	331
B Standardy używane w Ethereum	339

C	Kody operacji i zużycie paliwa w maszynie EVM w Ethereum	347
D	Narzędzia programistyczne, platformy i biblioteki	355
E	Biblioteka web3.js — samouczek	373
F	Krótkie odsyłacze	377
	Skorowidz	379

Podstawy Ethereum

W tym rozdziale rozpoczniemy poznawanie Ethereum. Dowiesz się, jak używać portfeli, jak tworzyć transakcje, a także jak uruchomić prosty inteligentny kontrakt.

Jednostki waluty ether

Jednostka waluty w Ethereum to *ether*. Inne stosowane nazwy to ETH, symbol Ξ (jest to grecka litera ksi; przypomina ona stylizowaną wielką literę E) lub, rzadziej, \blacklozenge . Oto przykłady: 1 ether, 1 ETH, Ξ 1 lub \blacklozenge 1.



Symbolowi Ξ odpowiada kod Unicode U+039E, a symbolowi \blacklozenge — kod Unicode U+2666.

Ether dzieli się na mniejsze jednostki, aż do najmniejszej, którą jest *wei*. Jeden ether to trylion wei (10^{18} , czyli 1 000 000 000 000 000 000). Możesz usłyszeć, jak niektóre osoby nazywają walutę „Ethereum”, jest to jednak częsty błąd nowicjuszy. Ethereum to system, natomiast walutą jest ether.

W Ethereum liczba etherów jest reprezentowana wewnętrznie zawsze jako liczba całkowita bez znaku reprezentująca wei. Gdy przesyłasz jeden ether, w transakcji jako wartość kodowany jest 1000000000000000000 wei.

Poszczególne jednostki waluty ether mają *nazwę naukową* zgodną z międzynarodowym układem jednostek miar *SI*, a także *nazwę potoczną*, honorującą liczne wybitne umysły ze świata informatyki i kryptografii.

W tabeli 2.1 wymieniono różne jednostki wraz z nazwami potocznymi i naukowymi. Aby zachować zgodność z wewnętrzną reprezentacją kwot, w tabeli wszystkie wartości są podane w wei (pierwszy wiersz), a ether przedstawiono jako 10^{18} wei (siódmy wiersz).

Tabela 2.1. Jednostki waluty ether i ich nazwy

Wartość (w wei)	Wykładnik	Nazwa potoczna	Nazwa w układzie SI
1		wei	Wei
1000	10^3	Babbage	Kilowei lub femtoether
1 000 000	10^6	Lovelace	Megawei lub picoether
1 000 000 000	10^9	Shannon	Gigawei lub nanoether
1 000 000 000 000	10^{12}	Szabo	Microether lub micro
1 000 000 000 000 000	10^{15}	Finney	Milliether lub milli
1 000 000 000 000 000 000	10^{18}	Ether	Ether
1 000 000 000 000 000 000 000	10^{21}	Grand	Kiloether
1 000 000 000 000 000 000 000 000	10^{24}		Megaether

Wybieranie portfela Ethereum

Pojęcie „portfel” może oznaczać wiele rzeczy, przy czym wszystkie te znaczenia są powiązane i w praktyce sprowadzają się do tego samego. Tu „portfel” oznacza aplikację pomagającą w zarządzaniu kontem Ethereum. Krótko mówiąc: portfel Ethereum to brama do systemu Ethereum. Zawiera klucze i pozwala tworzyć oraz rozsyłać transakcje w imieniu właściciela. Wybieranie portfela Ethereum bywa trudne, ponieważ dostępnych jest wiele różnych narzędzi o zróżnicowanych funkcjach i projektach. Niektóre portfele są lepsze dla początkujących, inne dla ekspertów. Sama platforma Ethereum wciąż jest usprawniana, a „najlepsze” portfele to zwykle te, które dostosowują się do zmian w platformie.

Nie martw się jednak. Jeśli po wybraniu portfela okaże się, że nie w pełni Ci on odpowiada lub że chcesz wypróbować inne narzędzie, możesz stosunkowo łatwo go zmienić. Wystarczy przeprowadzić transakcję przekazującą środki ze starego portfela do nowego lub wyeksportować klucze prywatne i zaimportować je w nowym portfelu.

W tej książce w przykładach używane będą trzy różne typy portfeli: mobilny, desktopowy i internetowy. Wybraliśmy te trzy rodzaje, ponieważ reprezentują one zróżnicowany poziom złożoności i funkcji. Nie wybraliśmy ich z powodu wysokiej jakości lub solidnych zabezpieczeń. Są one po prostu dobrymi narzędziami do testów i demonstracji rozwiązań.

Warto pamiętać, że aby portfel działał, musi mieć dostęp do kluczy prywatnych. Dlatego niezbędne jest, by pobierać i stosować wyłącznie portfele z zaufanych źródeł. Na szczęście zwykle jest tak, że im bardziej popularny jest dany portfel, tym większe można mieć do niego zaufanie. Jednak dobrą praktyką jest unikanie „wkładania wszystkich jajek do jednego koszyka”. Lepiej jest rozdzielić konto Ethereum między kilka portfeli.

Oto kilka dobrych portfeli na początek:

MetaMask

MetaMask to portfel będący rozszerzeniem przeglądarki. Działa on w przeglądarkach takich jak Chrome, Firefox, Opera i Brave Browser. Jest łatwy w użyciu i wygodny do wykonywania testów, ponieważ potrafi łączyć się z różnymi węzłami Ethereum i testowymi łańcuchami bloków. MetaMask jest portfelem internetowym.

Jaxx

Jaxx to portfel obsługujący wiele walut i platform. Działa w różnych systemach operacyjnych, takich jak Android, iOS, Windows, macOS i Linux. Często okazuje się dobrym wyborem dla początkujących, ponieważ został zaprojektowany z myślą o prostocie i łatwości użytkowania. Jaxx może być portfelem mobilnym i desktopowym (w zależności od tego, gdzie został zainstalowany).

MyEtherWallet (MEW)

MyEtherWallet to internetowy portfel działający w przeglądarce. Udostępnia wiele zaawansowanych funkcji, które poznasz w licznych przykładach z tej książki. MyEtherWallet jest portfelem internetowym.

Emerald Wallet

Emerald Wallet zaprojektowano na potrzeby łańcucha bloków Ethereum Classic, jednak obsługuje także inne łańcuchy bloków oparte na Ethereum. Jest to aplikacja desktopowa o otwartym dostępie do kodu źródłowego. Działa w systemach Windows, macOS i Linux. Emerald Wallet może uruchamiać pełny węzeł lub łączyć się ze zdalnym węzłem publicznym i pracować w trybie „lekkim”. Dostępne jest też powiązane narzędzie, które pozwala wykonywać wszystkie operacje w wierszu poleceń.

Zacniemy od zainstalowania portfela MetaMask na komputerze stacjonarnym. Najpierw jednak pokrótce omówimy kontrolowanie kluczy i zarządzanie nimi.

Kontrola i odpowiedzialność

Otwarte łańcuchy bloków takie jak Ethereum są ważne, ponieważ działają jako system **zdecentralizowany**. Wynika z tego wiele rzeczy, a jednym z bardzo ważnych aspektów jest to, że każdy użytkownik Ethereum może (i powinien) kontrolować własne klucze prywatne, ponieważ decydują one o dostępie do środków i inteligentnych kontraktów. Mechanizm dający dostęp do środków i inteligentnych kontraktów czasem nazywany jest „kontem” lub „portfelem”. Te pojęcia są w tym kontekście dość skomplikowane; szczegółowo opiszemy je dalej. Na razie zapamiętaj podstawową zasadę, zgodnie z którą jeden klucz prywatny równa się jednemu „kontu”. Niektórzy użytkownicy rezygnują z kontroli nad własnymi kluczami prywatnymi i korzystają z usług niezależnego nadzorca, np. giełdy internetowej. Z tej książki dowiesz się, jak przejąć kontrolę nad własnymi kluczami prywatnymi i jak nimi zarządzać.

Z kontrolą związana jest duża odpowiedzialność. Jeśli utracisz klucze prywatne, nie będziesz miał dostępu do środków i kontraktów. Nikt nie będzie mógł Ci pomóc w odzyskaniu dostępu — Twoje fundusze zostaną na zawsze zablokowane. Oto kilka wskazówek, które pomogą Ci udźwignąć tę odpowiedzialność:

- Nie eksperymentuj z bezpieczeństwem. Stosuj sprawdzone standardowe rozwiązania.
- Im ważniejsze konto (im wyższa wartość kontrolowanych środków lub istotniejszy dostępny inteligentny kontrakt), tym ściślejsze środki bezpieczeństwa należy zastosować.
- Najwyższy poziom bezpieczeństwa zapewniają urządzenia niepodłączone do sieci, jednak nie każde konto wymaga takich zabezpieczeń.

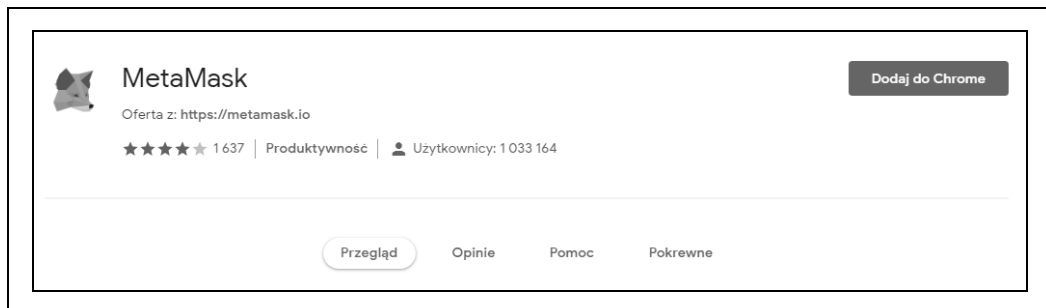
- Nigdy nie przechowuj kluczy prywatnej w czytelnej formie, zwłaszcza cyfrowo. Na szczęście większość dostępnych obecnie interfejsów użytkownika nie pozwoli Ci nawet zobaczyć kluczy prywatnych w nieprzetworzonej formie.
- Klucze prywatne można przechowywać w zaszyfrowanej postaci, takiej jak cyfrowy plik *keystore*. Ponieważ taki klucz jest zaszyfrowany, do jego odblokowania niezbędne jest hasło. Zadbaj o to, by było ono mocne (czyli długie i losowe), zarchiwizuj je i nie udostępniaj go. Jeśli nie używasz menedżera haseł, zapisz je i przechowuj w bezpiecznym, tajnym miejscu. Aby uzyskać dostęp do konta, potrzebny będzie plik *keystore* i hasło.
- Nie przechowuj haseł w dokumentach cyfrowych, na zdjęciach cyfrowych, zrzutach ekranu, dyskach sieciowych, w zaszyfrowanych plikach PDF itd. Nie eksperymentuj z bezpieczeństwem. Korzystaj z menedżera haseł lub z kartki i długopisu.
- Gdy chcesz zachować klucz jako mnemoniczną sekwencję słów, utwórz kopię fizyczną za pomocą kartki i długopisu. Nie odkładaj tego zadania „na później”, ponieważ możesz zapomnieć hasło. Taką kopię fizyczną możesz zastosować do odzyskania klucza prywatnego, gdy utracisz wszystkie dane zapisane w systemie lub zapomnisz hasło. Taka kopia może jednak zostać wykorzystana przez napastników do uzyskania Twoich kluczy prywatnych, dlatego nigdy nie zapisuj jej w formie cyfrowej. Ponadto przechowuj fizyczną kopię hasła schowaną bezpiecznie w zamkniętej szufladzie lub w sejfie.
- Przed przelewem dużej kwoty (zwłaszcza na nowy adres) najpierw wykonaj małą transakcję testową, np. o wartości złotówki, i poczekaj na jej potwierdzenie.
- Gdy stworzysz nowe konto, zacznij od przesłania na nowy adres tylko niewielkiej transakcji testowej. Gdy otrzymasz środki, spróbuj odesłać je z nowego konta. Jest wiele przyczyn, dla których tworzenie konta może się nie powieść. Lepiej dowiedzieć się o takim niepowodzeniu, przelewając niewielką kwotę. Jeśli testy zakończą się sukcesem, konto zostało poprawnie utworzone.
- Publiczne eksploratory bloków to łatwe narzędzie do samodzielnego sprawdzenia, czy transakcja została zaakceptowana w sieci. Ta wygoda ma jednak negatywny wpływ na prywatność, ponieważ ujawniasz swoje adresy eksploratorom bloków, co grozi tym, że będziesz śledzony.
- Nie przesyłaj środków na żaden z adresów podanych w tej książce. Klucze prywatne do tych adresów są podane w książce i ktoś natychmiast może przejąć Twoje pieniądze.

Po omówieniu podstawowych praktyk zarządzania kluczami i dbania o bezpieczeństwo pora przejść do używania portfela MetaMask!

Rozpoczynanie pracy z portfelem MetaMask

Otwórz przeglądarkę Google Chrome i przejdź pod adres <https://chrome.google.com/webstore/category/extensions>.

Poszukaj narzędzia MetaMask i kliknij logo z liskiem. Powinieneś zobaczyć informacje podobne do tych z rysunku 2.1.



Rysunek 2.1. Strona szczegółów rozszerzenia MetaMask Chrom

Ważne jest, abyś się upewnił, że pobrałeś prawdziwe rozszerzenie MetaMask. Czasem napastnikom udaje się „przemycić” szkodliwe rozszerzenie przez filtry Google’a. Oto dane prawdziwego rozszerzenia:

- ma identyfikator `nkbihfbeogaeaoehlfknkodbefgpgknn` na pasku adresu;
- jest udostępniane z witryny <https://metamask.io>;
- ma ponad 1400 recenzji;
- ma ponad 1 000 000 użytkowników.

Gdy się upewnisz, że wybrałeś poprawne rozszerzenie, kliknij przycisk *Dodaj do Chrome*, aby je zainstalować.

Tworzenie portfela

Po zainstalowaniu rozszerzenia MetaMask powinieneś zobaczyć nową ikonę (z głową liska) na pasku narzędzi przeglądarki. Kliknij tę ikonę, aby rozpocząć pracę. Pojawi się prośba o zaakceptowanie zasad i warunków. Następnie będziesz mógł utworzyć nowy portfel Ethereum. W tym celu wpisz hasło (zob. rysunek 2.2).

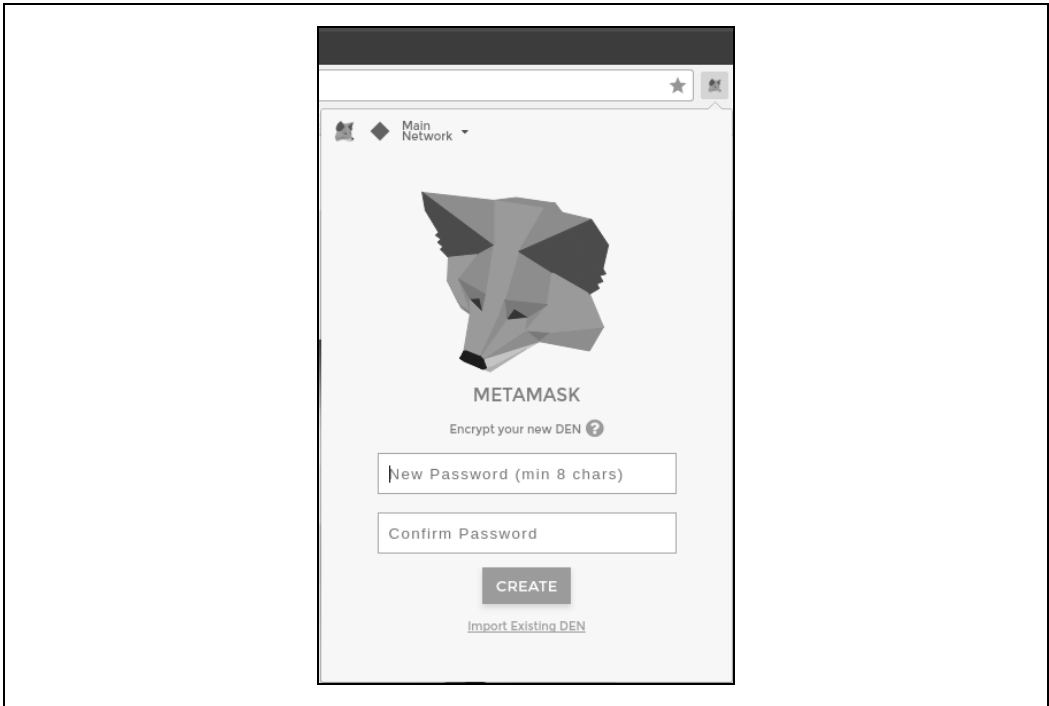


Dostęp do rozszerzenia MetaMask jest zabezpieczony hasłem, dlatego portfel nie może być stosowany przez dowolnego użytkownika przeglądarki.

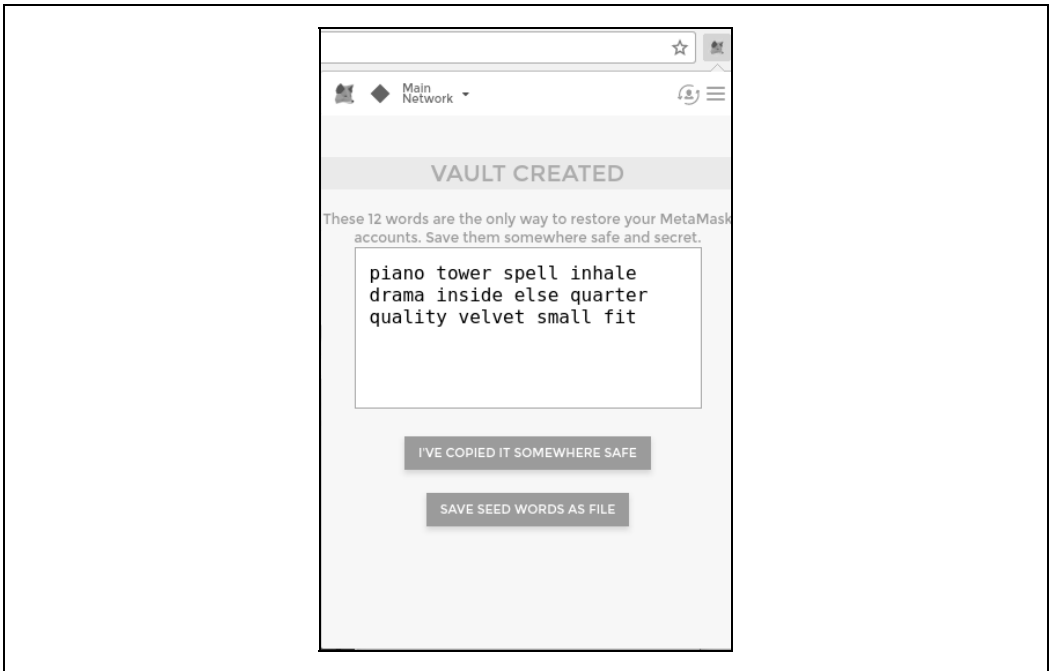
Po podaniu hasła MetaMask wygeneruje portfel i wyświetli **mnemoniczną kopię zapasową**, składającą się z 12 angielskich słów (zob. rysunek 2.3). Te słowa możesz zastosować w dowolnym zgodnym portfelu, aby odzyskać dostęp do środków, jeśli coś stanie się z rozszerzeniem MetaMask na Twoim komputerze. Do odzyskania konta nie będziesz potrzebował hasła — wystarczy 12 wygenerowanych słów.



Zapisz mnemoniczną kopię (12 słów) na kartce — i to dwukrotnie. Umieść obie kartki w dwóch odrębnych bezpiecznych miejscach, np. w ognioodpornym sejfie, zamykanej szufladzie lub skrytce depozytowej. Traktuj te kartki jak kwotę w gotówce odpowiadającą wartości środków w Twoim portfelu Ethereum. Każdy, kto uzyska dostęp do tych słów, będzie mógł ukraść Twoje pieniądze.

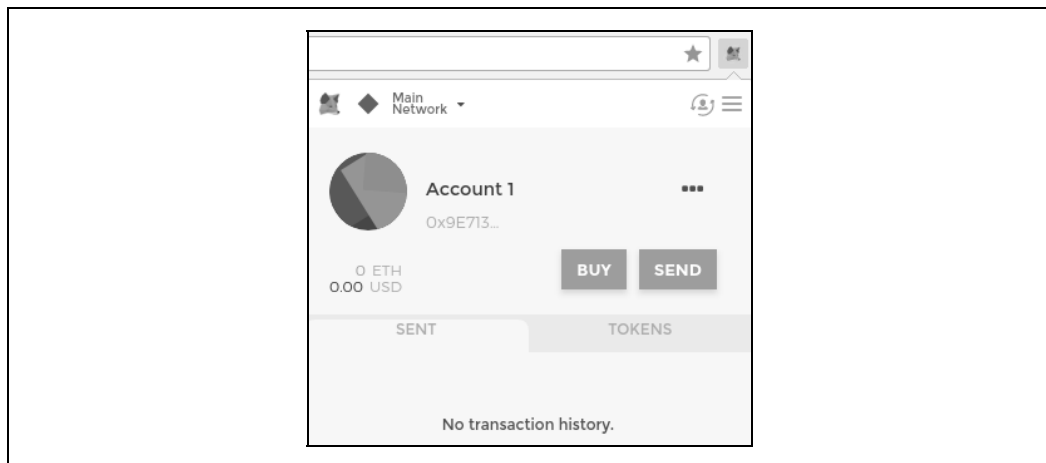


Rysunek 2.2. Strona na hasło w rozszerzeniu MetaMask dla przeglądarki Chrome



Rysunek 2.3. Mnemoniczna kopia zapasowa portfela wygenerowana przez rozszerzenie MetaMask

Po potwierdzeniu, że bezpiecznie zapisałeś mnemoniczną kopię zapasową, zobaczysz szczegółowe informacje na temat swojego konta Ethereum. Ilustruje to rysunek 2.4.



Rysunek 2.4. Konto Ethereum w rozszerzeniu MetaMask

Strona konta wyświetla jego nazwę (domyślnie jest to „Account 1”), adres Ethereum (w przykładzie jest to 0x9E713...) i kolorową ikonę, która pomaga odróżnić wizualnie dane konto od innych. W górnej części strony konta możesz zobaczyć, z jakiej sieci Ethereum aktualnie korzystasz (w przykładzie jest to sieć „Main Network”).

Gratulacje! Utworzyłeś swój pierwszy portfel Ethereum.

Zmienianie sieci

Na stronie konta w rozszerzeniu MetaMask widać, że możesz wybrać jedną z kilku sieci Ethereum. MetaMask domyślnie próbuje połączyć się z siecią główną (ang. *Main Network*). Inne możliwości to publiczne sieci testowe, dowolny wybrany węzeł Ethereum lub węzły udostępniające prywatne łańcuchy bloków na Twoim komputerze (na hoście lokalnym).

Główna sieć Ethereum

Jest to główny publiczny łańcuch bloków Ethereum — realne ethery, realna wartość i realne konsekwencje.

Sieć testowa Ropsten

Jest to publiczny testowy łańcuch bloków i publiczna testowa sieć Ethereum. Etery z tej sieci nie mają żadnej wartości.

Sieć testowa Kovan

Jest to publiczny testowy łańcuch bloków i publiczna testowa sieć Ethereum. Używany jest tu protokół osiągnięcia konsensusu Aura z modelem PoF (ang. *proof of authority*) i konsensusem federacyjnym. Etery z tej sieci nie mają żadnej wartości. Sieć testowa Kovan jest obsługiwana tylko w kliencie Parity. Inne klienty Ethereum do sprawdzania poprawności z użyciem modelu PoF używają zaproponowanego później protokołu osiągnięcia konsensusu Clique.

Sieć testowa Rinkeby

Jest to publiczny testowy łańcuch bloków i publiczna testowa sieć Ethereum. Używany jest tu protokół osiągnięcia konsensusu Clique z modelem PoF i konsensusem federacyjnym. Etery z tej sieci nie mają żadnej wartości.

Localhost 8545

Łączy się z węzłem działającym na tym samym komputerze, na którym pracuje przeglądarka. Ten węzeł może działać w dowolnym publicznym łańcuchu bloków (głównym lub testowym) albo w prywatnej sieci testowej.

Niestandardowy klient RPC

Umożliwia podłączenie rozszerzenia MetaMask do dowolnego węzła z interfejsem RPC zgodnym z klientem Geth. Taki węzeł może używać dowolnego publicznego lub prywatnego łańcucha bloków.

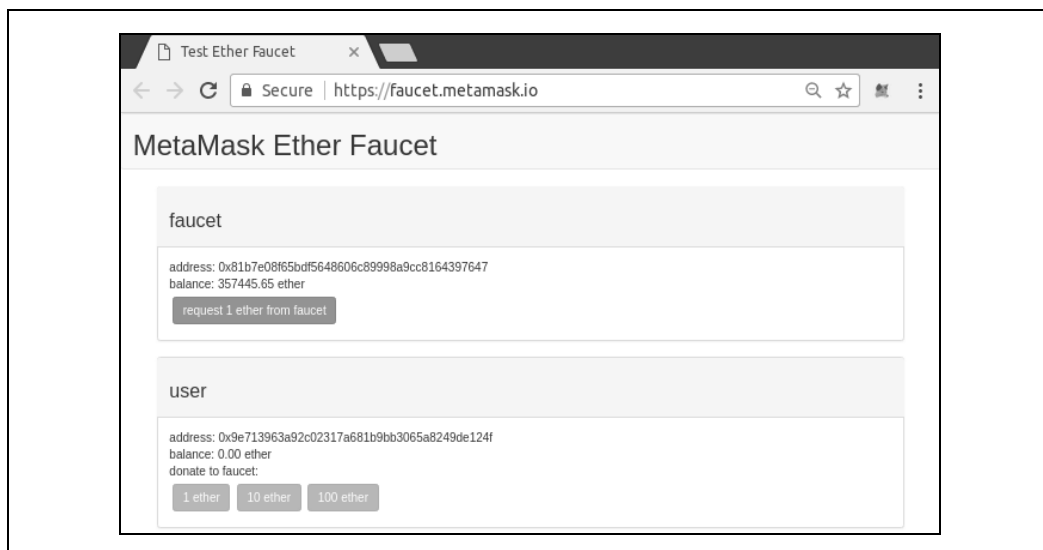


W portfelu MetaMask te same klucze prywatne i adresy Ethereum są używane we wszystkich sieciach, z jakimi portfel się łączy. Jednak w każdej z tych sieci stan konta dla adresu Ethereum będzie inny. Używane klucze mogą np. zapewniać dostęp do etherów i kontraktów w sieci Ropsten, ale już nie w sieci głównej.

Zdobywanie testowych etherów

Pierwsze zadanie polega na zapewnieniu środków w portfelu. Nie będziesz używał do tego sieci głównej, ponieważ realne ethery kosztują pieniądze, a zarządzanie nimi wymaga nieco doświadczenia. Na razie zasilisz portfel testowymi etherami.

Przełącz sieć w portfelu MetaMask na *Ropsten Test Network*. Kliknij przycisk *Buy*, a następnie wybierz opcję *Ropsten Test Faucet*. MetaMask otworzy nową stronę, widoczną na rysunku 2.5.



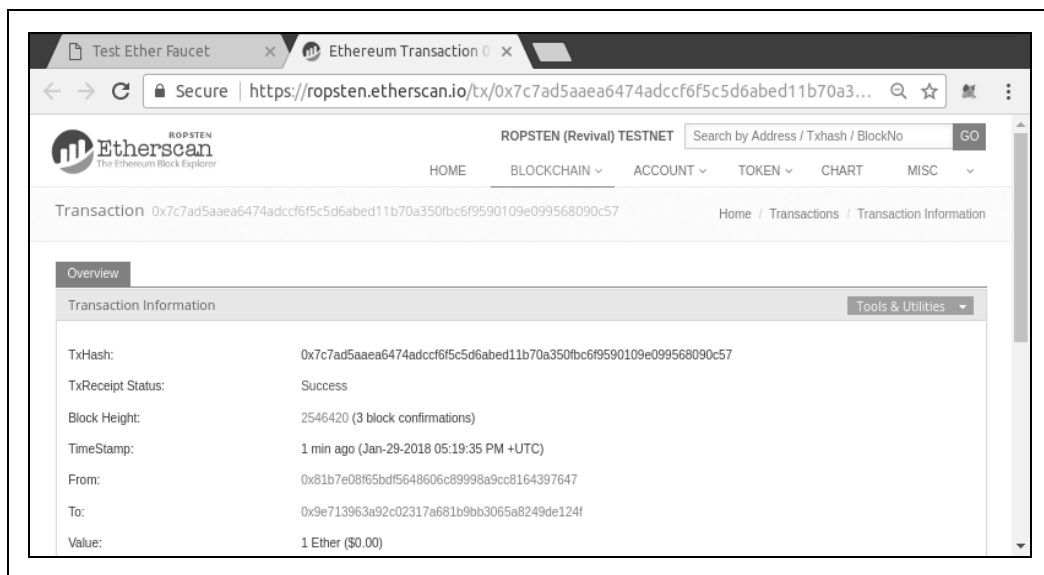
Rysunek 2.5. Kran z testowymi etherami w sieci Ropsten

Może zauważyłeś, że na stronie widoczny jest już adres Ethereum z Twojego portfela Ethereum. MetaMask integruje obsługujące Ethereum strony internetowe z portfelem MetaMask i wyświetla adresy Ethereum na takich stronach, co pozwala np. przesłać płatność do sklepu internetowego udostępniającego adres Ethereum. MetaMask może też podać na stronie internetowej Twój adres jako adres odbiorcy, jeśli strona tego zażąda. Na pokazanej stronie kran żąda od portfela MetaMask podania adresu, na który należy przesłać testowe ethery.

Kliknij zielony przycisk *request 1 ether from faucet*. W lewej dolnej części strony pojawi się identyfikator transakcji. Kran utworzył transakcję — płatność na Twoje konto. Oto identyfikator tej transakcji:

```
0x7c7ad5aaea6474adccf6f5c5d6abed11b70a350fbc6f9590109e099568090c57
```

Po kilku sekundach nowa transakcja zostanie wydobytą przez górników z sieci Ropsten, a w Twoim portfelu MetaMask pojawi się stan 1 ETH. Kliknij identyfikator transakcji, a przeglądarka otworzy **eksplorator bloków**, czyli witrynę, która pozwala wizualizować i przeglądać bloki, adresy oraz transakcje. MetaMask używa eksploratora bloków Etherscan (<https://etherscan.io/>). Jest to jeden z popularnych eksploratorów bloków Ethereum. Transakcja zawierająca przelew z kranu Ropsten Test Faucet jest pokazana na rysunku 2.6.



Rysunek 2.6. Eksplorator bloków Etherscan Ropsten

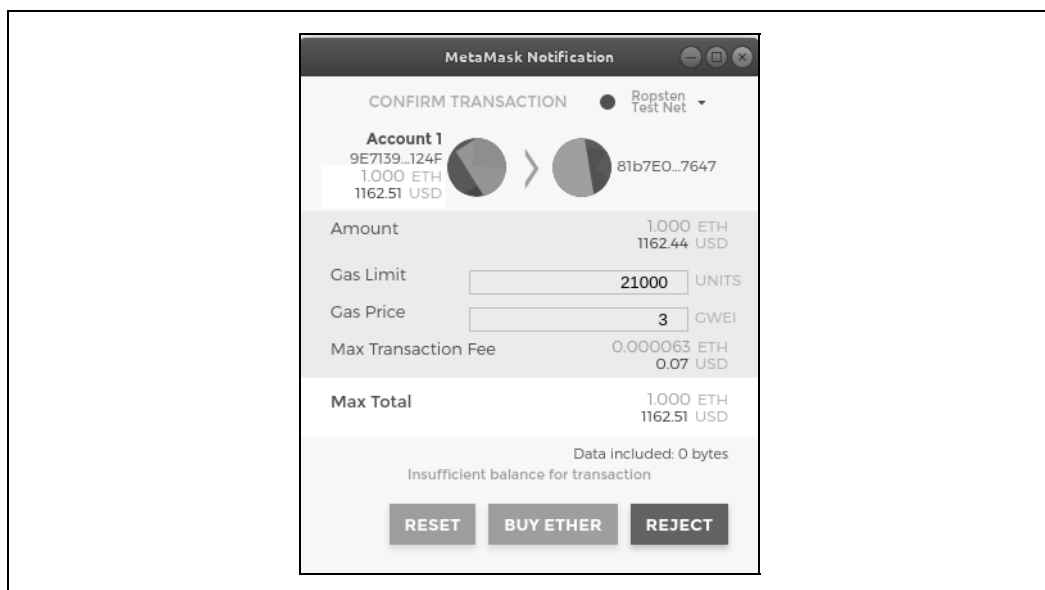
Ta transakcja została zarejestrowana w łańcuchu bloków Ropsten i może zostać wyświetlona w dowolnym momencie przez każdą osobę. Wystarczy poszukać identyfikatora transakcji lub kliknąć odsyłacz <http://bit.ly/2Q860Wk>.

Spróbuj kliknąć ten odsyłacz lub wpisz skrót transakcji w witrynie <http://ropsten.etherscan.io>, aby samemu zobaczyć dane.

Wysyłanie etherów z portfela MetaMask

Po otrzymaniu pierwszego testowego ethera z testowego kranu sieci Ropsten możesz zacząć eksperymenty z przesyłaniem etherów. Najpierw spróbuj przesłać środki z powrotem do kranu. Na stronie testowego kranu sieci Ropsten znajdziesz opcję przekazania 1 ETH do kranu. Ta opcja jest dostępna, abyś po zakończeniu testów mógł zwrócić resztę testowego ethera, co pozwoli wykorzystać te środki następną osobie. Choć testowe ethery nie mają żadnej wartości, niektóre osoby gromadzą je, co utrudnia innym korzystanie z sieci testowej. Gromadzenie testowych etherów jest niemile widziane!

Na szczęście nie jesteśmy zbieraczami testowych etherów. Kliknij pomarańczowy przycisk *1 ether*, aby nakazać portfelowi MetaMask utworzenie transakcji przekazującej 1 ether do kranu. MetaMask przygotowuje wtedy transakcję i wyświetli okno z prośbą o zatwierdzenie jej (rysunek 2.7).



Rysunek 2.7. Przesyłanie 1 ethera do kranu

Ups! Prawdopodobnie zauważyłeś, że nie możesz ukończyć transakcji. MetaMask informuje, że stan konta jest niewystarczający. Na pozór jest to dziwne. Masz 1 ETH i chcesz przesłać 1 ETH. Dlaczego więc MetaMask komunikuje, że nie masz wystarczających środków?

Powodem są koszty **paliwa**. Każda transakcja w Ethereum wymaga poniesienia opłat przekazywanych górnikom za sprawdzanie poprawności transakcji. W Ethereum opłaty są ponoszone w wirtualnej walucie nazywanej paliwem. Za paliwo płaci się etherami w ramach transakcji.



Opłaty są ponoszone także w sieciach testowych. Bez opłat sieć testowa działałaby inaczej od sieci głównej, przez co nie nadawałaby się na platformę testową. Ponadto opłaty chronią sieci testowe — tak samo jak i sieć główną — przed atakami DoS i źle skonstruowanymi kontraktami (np. z pętlami nieskończonymi).

Gdy przesyłaliśmy transakcję, MetaMask wyznaczył średnią cenę paliwa z ostatnich udanych transakcji na 3 gwei (czyli gigawei). Wei to najmniejsza jednostka waluty ether, co opisano w punkcie „Jednostki waluty ether”. Limit zużycia paliwa został tu ustawiony na koszt przesłania prostej transakcji i wynosi 21 000 jednostek paliwa. Tak więc maksymalny koszt w etherach wynosi $3 \cdot 21\,000 \text{ gwei} = 63\,000 \text{ gwei} = 0,000063 \text{ ETH}$. Pamiętaj, że ceny paliwa fluktuują, ponieważ są ustalane przede wszystkim przez górników. W dalszych rozdziałach zobaczysz, jak zwiększyć lub zmniejszyć limit zużycia paliwa, aby w razie potrzeby zagwarantować pierwszeństwo wykonania transakcji.

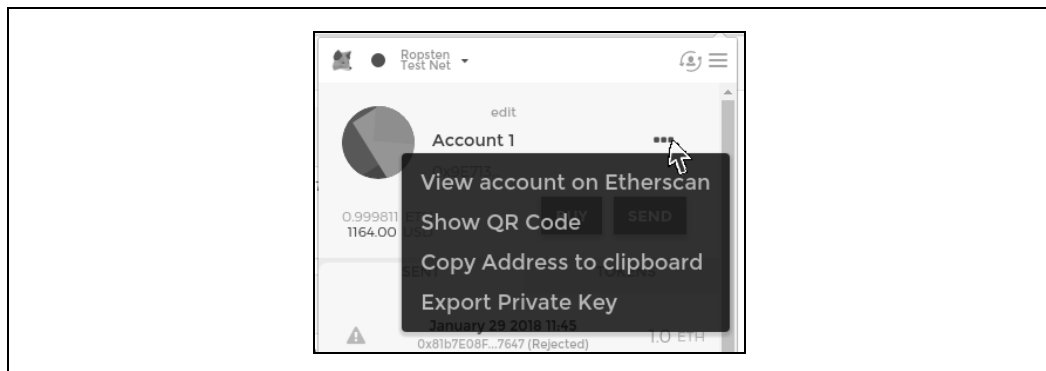
W podsumowaniu można napisać, że transakcja przelewu 1 ETH kosztuje 1,000063 ETH. MetaMask w mylący sposób zaokrągliła tę wartość *w dół* do 1 ETH, gdy wyświetla podsumowanie, jednak potrzebna kwota to 1,000063 ETH, a dostępny jest tylko 1 ETH. Kliknij przycisk *Reject*, aby anulować transakcję.

Pobierz więc więcej testowych etherów! Ponownie kliknij zielony przycisk *Request 1 ether from the faucet* i odczekaj kilka sekund. Nie martw się, kran udostępni dużo etherów przekaże Ci ich więcej, jeśli tego zażadasz.

Gdy stan konta wynosi 2 ETH, możesz ponowić próbę. Jeśli tym razem klikniesz pomarańczowy przycisk *1 ether* w celu przekazania środków, będziesz miał wystarczającą ilość środków na przeprowadzenie transakcji. Kiedy MetaMask wyświetli okno zatwierdzania płatności, kliknij przycisk *Submit*. Po tej operacji stan konta powinien wynosić 0,999937, ponieważ przesłałeś do kranu 1 ETH i poniosłeś koszty paliwa w wysokości 0,000063 ETH.

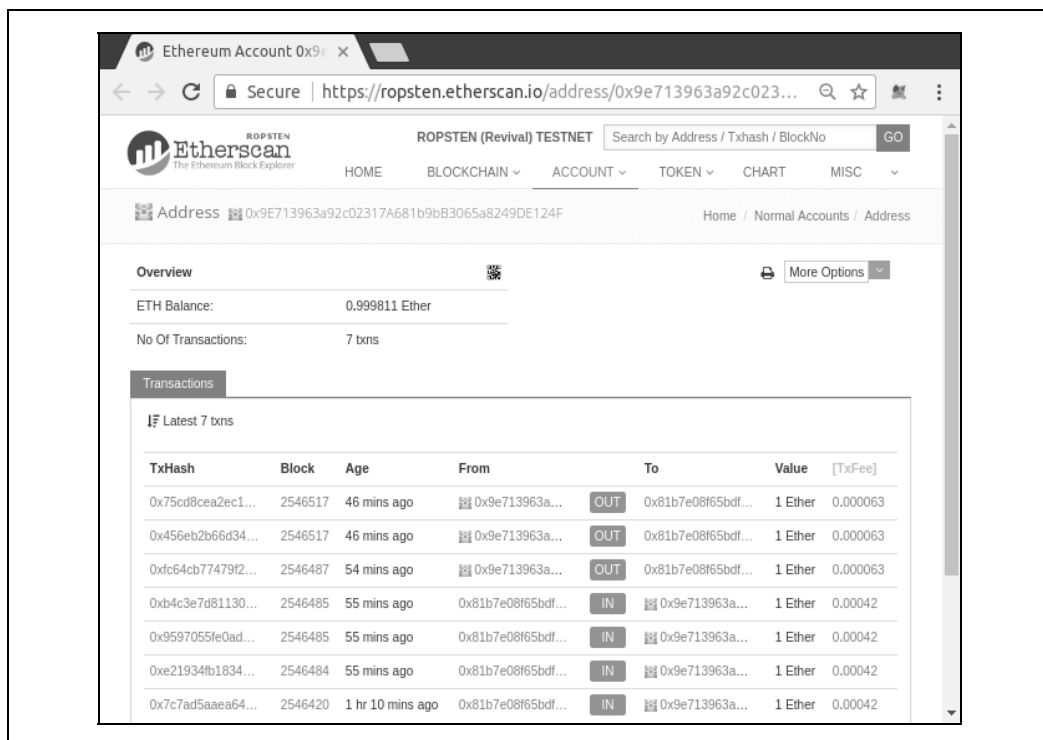
Przeglądanie historii transakcji dla adresu

Do tej pory stałeś się już ekspertem w używaniu portfela MetaMask do wysyłania i otrzymywania testowych etherów. Otrzymałeś już przynajmniej dwie płatności i wysłałeś co najmniej jedną. Wszystkie te transakcje możesz przejrzeć, używając eksploratora bloków <http://ropsten.etherscan.io>. Możesz albo skopiować adres portfela i wkleić go w polu wyszukiwania eksploratora bloków, albo otworzyć stronę za pośrednictwem portfela MetaMask. Obok ikony konta w portfelu MetaMask znajduje się przycisk z trzema kropkami. Kliknij go, aby wyświetlić menu opcji powiązanych z kontem (rysunek 2.8).



Rysunek 2.8. Menu kontekstowe konta w portfelu MetaMask

Wybierz opcję *View account on Etherscan*, aby otworzyć w eksploratorze bloków stronę internetową wyświetlającą historię transakcji na Twoim koncie (rysunek 2.9).



Rysunek 2.9. Historia transakcji dla adresu w eksploratorze Etherscan

W tym miejscu możesz zobaczyć całą historię operacji dla adresu Ethereum. Widoczne są tu wszystkie zarejestrowane w łańcuchu bloków Ropsten transakcje, w których Twój adres jest używany jako adres nadawcy lub odbiorcy. Kliknij kilka z tych transakcji, aby zobaczyć dodatkowe szczegóły.

Możesz zbadać historię transakcji dla dowolnego adresu. Przyjrzyj się historii transakcji dla adresu testowego kranu sieci Ropsten (wskazówka: jest to adres nadawcy w najstarszej płatności dla Twojego adresu). Zobaczysz wszystkie operacje przesłania testowych etherów z kranu do Ciebie i na inne adresy. Każda transakcja może prowadzić do kolejnych adresów i transakcji. Szybko zgubisz się w labiryncie powiązanych ze sobą danych. Publiczne łańcuchy bloków zawierają olbrzymią ilość informacji. Wszystkie te dane można analizować programowo, o czym przekonasz się dzięki lekturze dalszych rozdziałów.

Wprowadzenie do światowego komputera

Utworzyłeś już portfel oraz przesłałeś i otrzymałeś ethery. Do tej pory traktowaliśmy Ethereum jak kryptowalutę. Jednak Ethereum to coś o wiele więcej. Obsługa kryptowaluty jest drugorzędna względem podstawowej funkcji Ethereum, jaką jest udostępnianie zdecentralizowanego światowego

wego komputera. Ether ma służyć do opłaty za wykonywanie **inteligentnych kontraktów**, czyli programów komputerowych działających w emulowanym komputerze o nazwie *Ethereum Virtual Machine* (EVM).

Maszyna EVM jest singletonem w skali globalnej. Oznacza to, że działa tak, jakby była pojedynczym globalnym komputerem działającym wszędzie. Każdy węzeł w sieci Ethereum uruchamia lokalną kopię maszyny EVM, aby sprawdzać poprawność wykonywania kontraktu. Łańcuch bloków Ethereum rejestruje zmiany *stanu* światowego komputera, gdy ten przetwarza transakcje i inteligentne kontrakty. Te zagadnienia opisano szczegółowo w rozdziale 13.

Konta EOA i konta kontraktów

Portfel MetaMask tworzy konta typu EOA (ang. *externally owned account*). Takie konta charakteryzują się posiadaniem kluczy prywatnych. Klucze prywatne zapewniają kontrolę nad środkami i kontraktami. Prawdopodobnie domyślasz się już, że istnieje też inny typ kont. Są to **konta kontraktów**. Konto kontraktu to kod inteligentnego kontraktu, którego proste konta EOA nie mogą zawierać. Konto kontraktu nie ma też klucza prywatnego. Zamiast tego jego „właścicielem” (i zarządcą) jest logika kodu inteligentnego kontraktu — programu zapisanego w łańcuchu bloków Ethereum w momencie tworzenia konta kontraktu i wykonywanego przez maszynę EVM.

Kontrakty, podobnie jak konta EOA, mają adresy. Kontrakty, też tak jak konta EOA, mogą wysyłać i przyjmować ethery. Jednak gdy docelowym adresem transakcji jest adres kontraktu, powoduje to *uruchomienie* kontraktu w maszynie EVM z użyciem określonej transakcji i jej danych jako danych wejściowych. Obok etherów transakcje mogą też obejmować *dane* określające, którą funkcję z kontraktu należy uruchomić i jakie parametry do niej przekazać. W ten sposób transakcje mogą *wywoływać* funkcje w kontraktach.

Warto zauważyć, że ponieważ konto kontraktu nie ma klucza prywatnego, nie może *zainicjować* transakcji. Tylko konta EOA mogą inicjować transakcje. Kontrakty mogą jednak *reagować* na transakcje, wywołując inne kontrakty. Powstają w ten sposób złożone ścieżki wykonywania kodu. Jednym z typowych zastosowań tej techniki jest wysyłanie przez kontrakt EOA żądania transakcji do portfela z obsługą inteligentnych kontraktów i wielopodpisu, aby przesłać ethery na inny adres. Typowy wzorzec programowania aplikacji DApp polega na tym, że kontrakt A kieruje wywołanie do kontraktu B, aby zachować wspólny stan wśród użytkowników kontraktu A.

W kilku następujących punktach zobaczysz, jak napisać pierwszy kontrakt. Następnie dowiesz się, jak utworzyć, zasilić i zastosować ten kontrakt za pomocą portfela MetaMask i testowych etherów w sieci testowej Ropsten.

Prosty kontrakt — kran z testowymi etherami

Ethereum obsługuje wiele różnych wysokopoziomowych języków. Wszystkie te języki można stosować do pisania kontraktów i tworzenia kodu bajtowego dla maszyny EVM. O wybranych najważniejszych i najciekawszych językach przeczytasz w punkcie „Wprowadzenie do wysokopoziomowych języków dostępnych w Ethereum”. Do programowania inteligentnych kontraktów zdecydowanie

najczęściej używany jest jeden z wysokopoziomowych języków — Solidity. Został on opracowany przez dr. Gavina Wooda, współautora tej książki, i stał się najczęściej używanym językiem w Ethereum (i nie tylko tu). Do napisania pierwszego kontraktu posłużysz się właśnie tym językiem.

W pierwszym przykładzie (listing 2.1) napiszesz kontrakt kontrolujący **kran**. Używałeś już kranu, aby otrzymać testowe ethery w sieci Ropsten. Kran to stosunkowo prosty mechanizm — przekazuje ethery na podany adres i może być okresowo uzupełniany środkami. Kran możesz zaimplementować jako portfel kontrolowany przez człowieka lub serwer WWW.

Listing 2.1. Plik *Faucet.sol* — napisany w Solidity kontrakt działający jak kran

```
1 // Pierwszy kontrakt działa jak kran.
2 contract Faucet {
3
4     // Przekazuje ethery na dowolny wskazany adres.
5     function withdraw(uint withdraw_amount) public {
6
7         // Ograniczanie żądanej kwoty.
8         require(withdraw_amount <= 1000000000000000000);
9
10        // Przekazywanie środków na wskazany adres.
11        msg.sender.transfer(withdraw_amount);
12    }
13
14    // Przyjmuje dowolną przesyłaną kwotę.
15    function () public payable {}
16
17 }
```



Wszystkie przykładowe programy do tej książki znajdziesz w podkatalogu *code* w repozytorium GitHub powiązanej z tą pozycją (<https://github.com/ethereumbook/ethereumbook/>). Kontrakt *Faucet.sol* znajdziesz w katalogu *code/Solidity/Faucet.sol*. Spolszczona wersja kodu jest dostępna w witrynie wydawnictwa Helion.

Jest to bardzo prosty kontrakt — prawie tak prosty, jak to możliwe. Ma on pewne *wady*, aby zilustrować niezalecane praktyki i luki bezpieczeństwa. W dalszych punktach będziesz się uczył, analizując błędy z tego programu. Na razie przyjrzyj się wiersz po wierszu temu, co ten kontrakt robi i jak działa. Szybko zauważysz, że liczne elementy języka Solidity są podobne jak w innych językach programowania, takich jak JavaScript, Java i C++.

Pierwszy wiersz to komentarz:

```
// Pierwszy kontrakt działa jak kran.
```

Komentarze są przeznaczone dla ludzi i nie znajdują się w wykonywalnym kodzie bajtowym dla maszyny EVM. Zwykle umieszcza się je w wierszu przed objaśnianym kodem (a czasem w tym samym wierszu). Komentarze rozpoczynają się od dwóch ukośników — *//*. Wszystko od pierwszego ukośnika to końca wiersza jest traktowane jak pusty wiersz i pomijane.

W następnym wierszu rozpoczyna się właściwy kontrakt:

```
contract Faucet {
```

W tym wierszu zadeklarowany jest kontrakt (`contract`); przypomina on klasy (deklaracje `class`) z innych języków obiektowych. Definicja kontraktu obejmuje wszystkie wiersze między nawiasami klamrowymi `{}`, definiującymi **zasięg** (podobną funkcję nawiasy klamrowe pełnią także w wielu innych językach programowania).

Dalej zadeklarowana jest pierwsza funkcja kontraktu `Faucet`:

```
function withdraw(uint withdraw_amount) public {
```

Ta funkcja nosi nazwę `withdraw` i przyjmuje jeden argument, `withdraw_amount`, typu `uint` (jest to liczba całkowita bez znaku). Omawiana funkcja jest zadeklarowana jako publiczna, co oznacza, że może być wywoływana przez inne kontrakty. Dalej, w nawiasach klamrowych, znajduje się definicja tej funkcji. Pierwszy fragment funkcji `withdraw` określa limit wypłat:

```
require(withdraw_amount <= 100000000000000000);
```

Używana jest tu wbudowana funkcja Solidity `require`, aby sprawdzać warunek, zgodnie z którym parametr `withdraw_amount` ma wartość nie większą niż 100 000 000 000 000 000 wei (wei to podstawowa jednostka waluty ether; zob. tabelę 2.1), czyli 0,1 ethera. Jeśli funkcja `withdraw` zostanie wywołana z parametrem `withdraw_amount` przekraczającym podaną wartość, funkcja `require` zatrzyma wykonywanie kontraktu i zgłosi *wyjatek*. Warto zauważyć, że w Solidity instrukcje muszą kończyć się średnikiem.

Ta część kontraktu określa logikę działania kranu. Ten kod kontroluje przekazywanie środków przez kontrakt, ponieważ nakłada limit na wypłaty. Jest to bardzo prosty mechanizm, jednak daje przedsmak możliwości programowalnych łańcuchów bloków w zakresie tworzenia zdecentralizowanego oprogramowania do kontroli pieniędzy.

Dalej znajduje się kod wypłacający środki:

```
msg.sender.transfer(withdraw_amount);
```

Dzieje się tu kilka ciekawych rzeczy. Obiekt `msg` to jeden z obiektów dostępnych we wszystkich kontraktach. Reprezentuje on transakcję, która uruchomiła wykonywanie danego kontraktu. Atrybut `sender` to adres nadawcy transakcji. Funkcja `transfer` to wbudowana funkcja przekazująca ethery z bieżącego kontraktu na adres nadawcy. Przeczytajmy tę instrukcję od końca: oznacza ona przesyłanie (`transfer`) środków do nadawcy (`sender`) komunikatu (`msg`), który uruchomił wykonywanie kontraktu. Funkcja `transfer` przyjmuje jeden argument — kwotę. Jako ten argument przekazywana jest wartość parametru `withdraw_amount` z zadeklarowanej kilka wierszy wcześniej funkcji `withdraw`.

Kolejny wiersz to zamykający nawias klamrowy. Oznacza on koniec definicji funkcji `withdraw`.

Dalej zadeklarowana jest kolejna funkcja:

```
function () public payable {}
```

Jest to funkcja **rezerwowa** lub **domyślna**. Jest ona wywoływana, jeśli w transakcji, która uruchomiła kontrakt, albo nie podano nazwy żadnej funkcji zadeklarowanej w kontrakcie, albo w ogóle nie wywołano żadnej funkcji, albo w ogóle nie umieszczono danych. Kontrakty mogą mieć jedną funkcję domyślną (bez nazwy) i zwykle służy ona do przyjmowania etherów. To dlatego została zdefiniowana jako publiczna i z obsługą płatności (specyfikator `payable`), co oznacza, że kontrakt

może pobierać ethery. Funkcja ta nie robi nic oprócz przyjmowania etherów. Wskazuje na to pusta definicja w nawiasach klamrowych. Jeśli utworzysz transakcję, która przesyła ethery na adres kontraktu tak, jakby był to portfel, ta funkcja obsłuży tę operację.

Pod funkcją domyślną znajduje się ostatni zamykający nawias klamrowy. Kończy on definicję kontraktu Faucet. To wszystko!

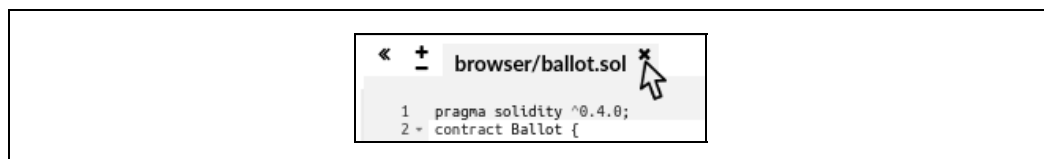
Kompilowanie kontraktu Faucet

Po utworzeniu pierwszego przykładowego kontraktu należy posłużyć się kompilatorem języka Solidity, aby przekształcić kod w języku Solidity na kod bajtowy maszyny EVM, aby można go wykonywać w maszynie EVM w łańcuchu bloków.

Kompilator języka Solidity to niezależny plik wykonywalny dostępny w różnych platformach i środowiskach IDE. Aby uprościć pracę, użyjemy jednego z popularnych środowisk IDE — **Remix**.

W przeglądarce Chrome (z zainstalowanym wcześniej portfelem MetaMask) przejdź do środowiska IDE Remix na stronie <https://remix.ethereum.org>.

Kiedy po raz pierwszy otworzysz środowisko Remix, widoczny będzie przykładowy kontrakt *ballot.sol*. Nie jest on potrzebny, dlatego możesz go zamknąć. Kliknij w tym celu symbol „x” w rogu zakładki (rysunek 2.10).



Rysunek 2.10. Zamykanie zakładki z domyślnym przykładowym kodem

Następnie dodaj nową zakładkę. W tym celu kliknij okrągły znak plus w pasku narzędzi w lewym górnym rogu (rysunek 2.11). Nowy plik nazwij *Faucet.sol*.



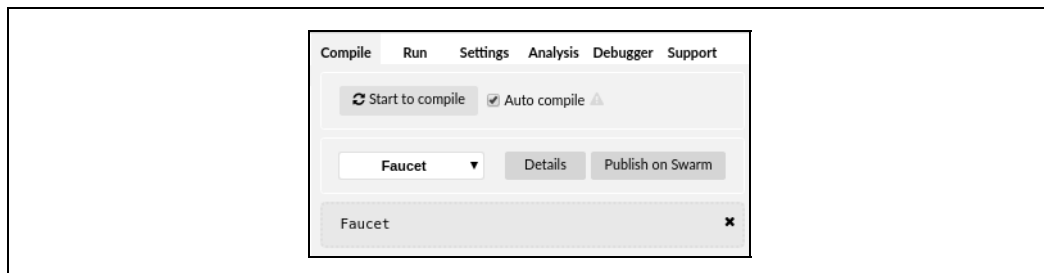
Rysunek 2.11. Kliknij znak plus, aby otworzyć nową zakładkę

Po otwarciu nowej zakładki skopiuj i wklej kod z przykładowego pliku *Faucet.sol* (rysunek 2.12).

Po wczytaniu kontraktu *Faucet.sol* do środowiska IDE Remix środowisko automatycznie skompiluje kod. Jeśli wszystko przebiegnie prawidłowo, po prawej stronie pod zakładką *Compile* pojawi się zielone pole z napisem *Faucet*. Jest to potwierdzenie udanej kompilacji (rysunek 2.13).

```
browser/Faucet.sol x ContractDefinition test3 ↗
1 // Wersja kompilatora Solidity, dla której napisano program.
2 pragma solidity ^0.4.19;
3
4 // Pierwszy kontrakt działa jak kran.
5 contract Faucet {
6
7 // Przekazuje ethery na dowolny wskazany adres.
8 function withdraw(uint withdraw_amount) public {
9
```

Rysunek 2.12. Skopiuj do nowej zakładki kod przykładowego kontraktu Faucet



Rysunek 2.13. Remix z powodzeniem kompiluje kontrakt Faucet.sol

Jeśli coś się nie powiodło, najbardziej prawdopodobnym problemem jest to, że środowisko IDE Remix używa kompilatora Solidity w wersji innej niż 0.4.19. Wtedy dyrektywa `pragma` sprawi, że kontrakt `Faucet.sol` się nie skompiluje. Aby zmienić wersję kompilatora, otwórz zakładkę `Settings`, wybierz wersję 0.4.19 i ponów próbę¹.

Kompilator języka Solidity skompilował plik `Faucet.sol` do postaci kodu bajtowego dla maszyny EVM. Jeśli jesteś ciekawy, ten kod bajtowy wygląda tak:

```
PUSH1 0x60 PUSH1 0x40 MSTORE CALLVALUE ISZERO PUSH2 0xF JUMPI PUSH1 0x0 DUP1
REVERT JUMPDEST PUSH1 0xE5 DUP1 PUSH2 0x1D PUSH1 0x0 CODECOPY PUSH1 0x0 RETURN
STOP PUSH1 0x60 PUSH1 0x40 MSTORE PUSH1 0x4 CALLDATASIZE LT PUSH1 0x3F JUMPI
PUSH1 0x0 CALLDATALOAD PUSH29
0x1000000000000000000000000000000000000000000000000000000000000000
SWAP1 DIV PUSH4 0xFFFFFFFF AND DUP1 PUSH4 0x2E1A7D4D EQ PUSH1 0x41 JUMPI
JUMPDEST STOP JUMPDEST CALLVALUE ISZERO PUSH1 0x4B JUMPI PUSH1 0x0 DUP1 REVERT
JUMPDEST PUSH1 0x5F PUSH1 0x4 DUP1 DUP1 CALLDATALOAD SWAP1 PUSH1 0x20 ADD SWAP1
SWAP2 SWAP1 POP POP PUSH1 0x61 JUMP JUMPDEST STOP JUMPDEST PUSH8
0x16345785D8A0000 DUP2 GT ISZERO ISZERO ISZERO PUSH1 0x77 JUMPI PUSH1 0x0 DUP1
REVERT JUMPDEST CALLER PUSH20 0xFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF
AND
PUSH2 0x8FC DUP3 SWAP1 DUP2 ISZERO MUL SWAP1 PUSH1 0x40 MLOAD PUSH1 0x0 PUSH1
0x40 MLOAD DUP1 DUP4 SUB DUP2 DUP6 DUP9 DUP9 CALL SWAP4 POP POP POP ISZERO
ISZERO PUSH1 0xB6 JUMPI PUSH1 0x0 DUP1 REVERT JUMPDEST POP JUMP STOP LOG1 PUSH6
0x627A7A723058 KECCAK256 PUSH9 0x13D1EA839A4438EF75 GASLIMIT CALLVALUE LOG4 0x5f
PUSH24 0x7541F409787592C988A079407FB28B4AD00029000000000000
```

Czy nie jesteś zadowolony, że używasz wysokopoziomowego języka takiego jak Solidity, zamiast programować bezpośrednio w kodzie bajtowym dla maszyny EVM? Ja jestem!

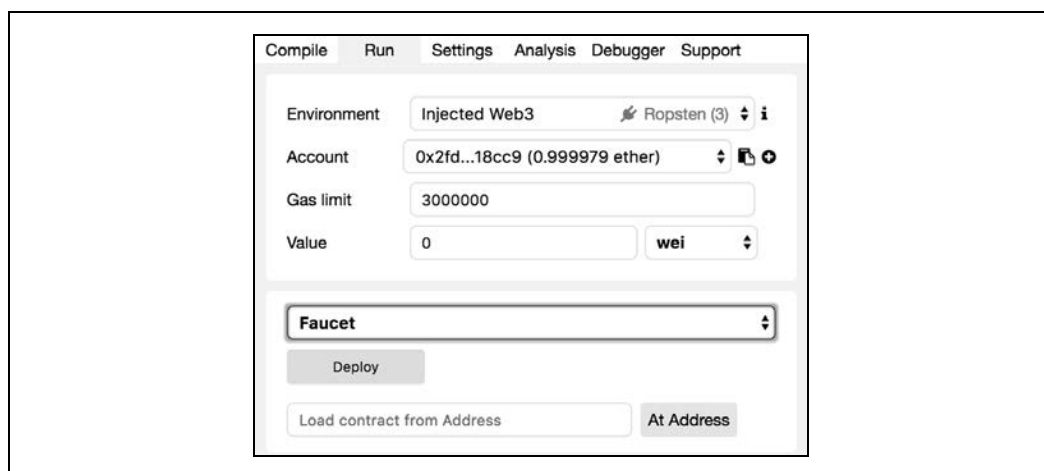
¹ Obecnie wersję kompilatora można zmienić w zakładce `Compile` — *przyp. tłum.*

Tworzenie kontraktu w łańcuchu bloków

Mamy więc kontrakt skompilowany do postaci kodu bajtowego. Teraz trzeba „zarejestrować” go w łańcuchu bloków Ethereum. Do przetestowania kontraktu posłużą sieć testowa Ropsten, dlatego trzeba przesłać kontrakt do łańcucha bloków Ropsten.

Rejestrowanie kontraktu w łańcuchu bloków wymaga utworzenia specjalnej transakcji z adresem docelowym `0x00` (jest to **adres zerowy**). Jest to specjalny adres, informujący łańcuch bloków Ethereum, że chcesz zarejestrować kontrakt. Na szczęście środowisko IDE Remix wykona wszystkie potrzebne operacje i prześle transakcję do portfela MetaMask.

Najpierw otwórz zakładkę *Run* i wybierz opcję *Injected Web3* na liście rozwijanej *Environment*. Środowisko IDE Remix połączy się wtedy z portfelem MetaMask, a za pośrednictwem tego portfela — także z siecią testową Ropsten. Zobaczysz wtedy nazwę *Ropsten* przy polu *Environment*. W polu *Account* widoczny jest adres używanego portfela (rysunek 2.14).



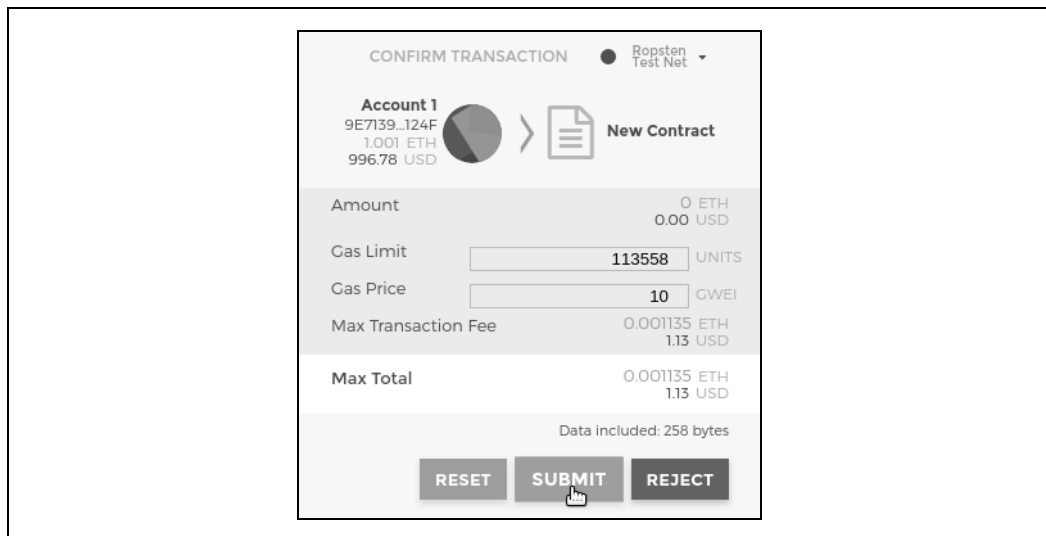
Rysunek 2.14. Zakładka *Run* w środowisku IDE Remix. Wybrane jest środowisko *Injected Web3*

Pod wybranymi właśnie ustawieniami uruchamiania kodu znajduje się gotowy do utworzenia kontrakt *Faucet*. Kliknij widoczny na rysunku 2.14 przycisk *Deploy*.

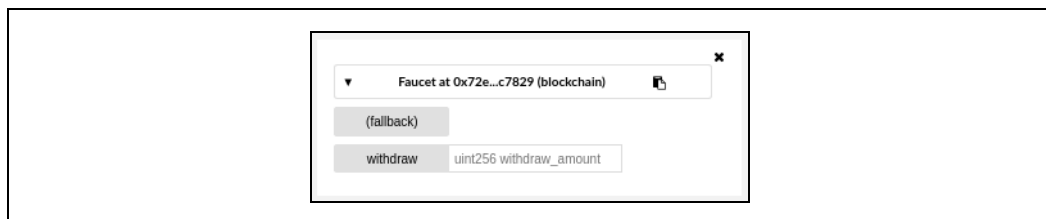
Remix utworzy wtedy specjalną transakcję tworzącą kontrakt, a MetaMask wyświetli prośbę o jej zatwierdzenie (rysunek 2.15). Warto zauważyć, że w tej transakcji tworzącej kontrakt nie są używane ethery, natomiast obejmuje ona 258 bajtów danych (jest to skompilowany kontrakt). Transakcja ta zużyje paliwo o wartości 10 gwei. Kliknij przycisk *Submit*, aby zatwierdzić transakcję.

Teraz musisz poczekać. Wydobycie kontraktu w sieci Ropsten zajmuje ok. 15 – 30 s. Środowisko Remix będzie wydawać się nieaktywne, zachowaj jednak cierpliwość.

Po utworzeniu kontrakt będzie widoczny w dolnej części zakładki *Run* (rysunek 2.16).



Rysunek 2.15. MetaMask wyświetla transakcję tworzącą kontrakt



Rysunek 2.16. Kontrakt Faucet jest AKTYWNY!

Warto zauważyć, że kontrakt Faucet ma teraz własny adres. Remix wyświetla go w napisie *Faucet at 0x72e...c7829* (Twój adres w postaci losowych liter i cyfr będzie inny). Niewielka ikona schowka widoczna po prawej stronie umożliwia skopiowanie adresu kontraktu do schowka. Posłużymy się tym adresem w następnym podrozdziale.

Interakcja z kontraktem

Warto podsumować przedstawione do tego miejsca informacje: kontrakty w Ethereum to programy kontrolujące pieniądze i działające w maszynie wirtualnej EVM. Są tworzone za pomocą specjalnej transakcji, która przesyła kod bajtowy rejestrowany w łańcuchu bloków. Kontrakt utworzony w łańcuchu bloków ma (podobnie jak portfele) adres Ethereum. Za każdym razem, gdy ktoś prześle transakcję na adres kontraktu, spowoduje to uruchomienie kontraktu w maszynie EVM z określoną transakcją jako danymi wejściowymi. Transakcje przesyłane na adres kontraktu mogą zawierać ethery, dane lub oba te komponenty. Jeśli transakcja zawiera ethery, są one deponowane w stanie konta. Jeżeli transakcja obejmuje dane, mogą one określać funkcję z kontraktu i wywoływać ją z podanymi argumentami.

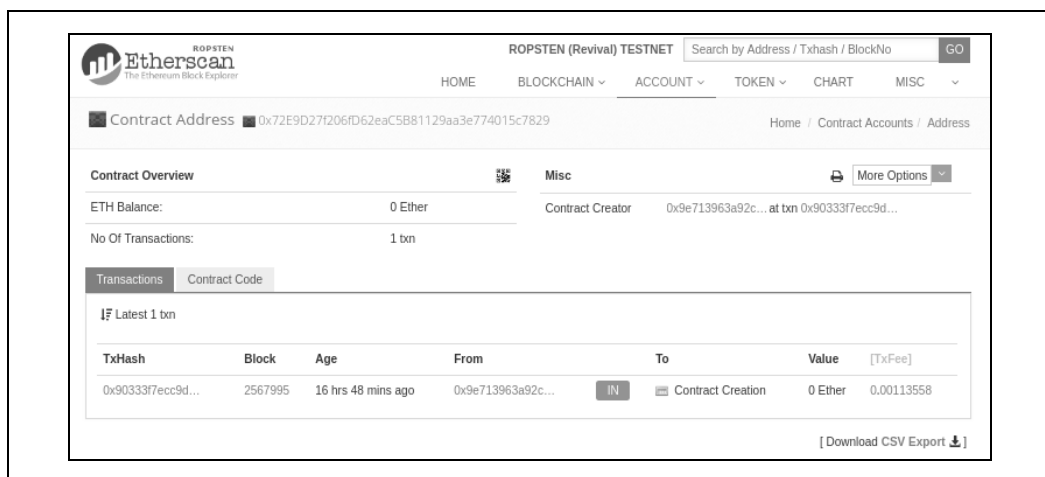
Wyświetlanie adresu kontraktu w eksploratorze bloków

Kontrakt został już zarejestrowany w łańcuchu bloków i można się przekonać, że ma on adres Ethereum. Otwórz eksplorator bloków <https://ropsten.etherscan.io/> i sprawdź, jak wygląda kontrakt. W środowisku IDE Remix skopiuj adres kontraktu. W tym celu kliknij ikonę schowka widoczną obok nazwy kontraktu (rysunek 2.17).



Rysunek 2.17. Kopiowanie adresu kontraktu ze środowiska Remix

Pozostaw środowisko Remix otwarte. Wrócisz do niego później. Teraz przejdź w przeglądarce na stronę <https://ropsten.etherscan.io/> i wklej adres kontraktu w polu wyszukiwania. Powinieneś zobaczyć historię dla przypisanego do kontraktu adresu Ethereum (rysunek 2.18).



Rysunek 2.18. Wyświetlanie adresu kontraktu Faucet w eksploratorze bloków Etherscan

Zasilanie kontraktu

Na razie historia kontraktu obejmuje tylko jedną transakcję, tworzącą ten kontrakt. W eksploratorze widać, że kontrakt nie ma dostępnych etherów (stan jest równy zero). Wynika to z tego, że nie przesłano żadnych etherów do kontraktu w tworzącej go transakcji, choć było to możliwe.

Utworzony kran wymaga środków! Pierwszym zadaniem będzie użycie portfela MetaMask do przesłania etherów do kontraktu. Nadal powinieneś mieć dostępny w schowku adres kontraktu

(w przeciwnym razie ponownie skopiuj ten adres ze środowiska Remix). Otwórz portfel MetaMask i prześlij 1 ether na adres kontraktu w taki sam sposób, jakbyś używał dowolnego innego adresu Ethereum (rysunek 2.19).



Rysunek 2.19. Prześlij 1 ether na adres kontraktu

Jeśli po chwili odświeżysz stronę z eksploratorem bloków Etherscan, zobaczysz następną transakcję powiązaną z adresem kontraktu i zaktualizowany stan na poziomie 1 ethera.

Pamiętasz anonimową domyślną funkcję publiczną typu payable z kodu kontraktu *Faucet.sol*? Wygląda ona tak:

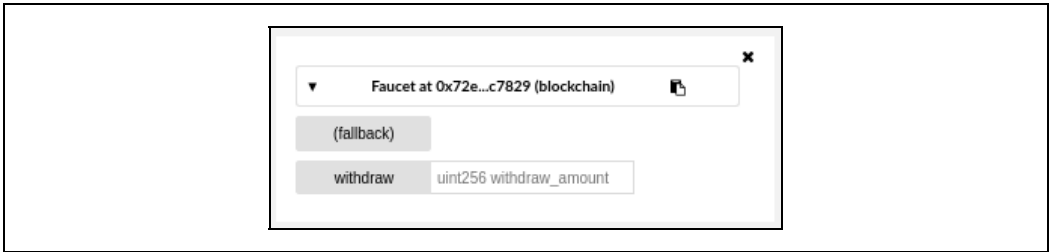
```
function () public payable {}
```

Gdy na adres kontaktu przesłałeś transakcję bez danych określających, którą funkcję należy wywołać, uruchomiona została ta domyślna funkcja. Ponieważ została ona zadeklarowana jako payable, przyjęła i zdeponowała 1 ether na koncie kontraktu. Transakcja spowodowała uruchomienie kontraktu w maszynie EVM i zaktualizowanie jego stanu. Właśnie zasililiś swój kran!

Wycofywanie środków z kontraktu

Następnie pobierzmy środki z kranu. W tym celu trzeba utworzyć transakcję, która wywołuje funkcję `withdraw` i przekazuje do niej argument `withdraw_amount`. Aby uprościć na razie rozwiązanie, użyjemy środowiska Remix do utworzenia transakcji, a portfel MetaMask wyświetli ją do zatwierdzenia.

Ponownie otwórz środowisko Remix i przyjrzyj się kontraktowi z zakładki *Run*. Powinieneś zobaczyć czerwone pole z etykietą `withdraw` i pole tekstowe z napisem `uint256 withdraw_amount` (rysunek 2.20).



Rysunek 2.20. Funkcja `withdraw` kontraktu `Faucet.sol` w środowisku Remix

Jest to interfejs kontraktu w środowisku Remix. Ten interfejs umożliwia tworzenie transakcji, które wywołują funkcje zdefiniowane w kontrakcie. Możesz wpisać wartość parametru `withdraw_amount` i kliknąć przycisk `withdraw`, aby wygenerować transakcję.

Najpierw należy ustalić wartość parametru `withdraw_amount`. Chcemy pobrać 0,1 ethera, co jest maksymalną kwotą dopuszczalną w kontrakcie. Pamiętaj, że wszystkie wartości w Ethereum są wewnętrznie zapisywane w jednostkach wei. Funkcja `withdraw` także oczekuje, że parametr `withdraw_amount` będzie liczbą wei. Pożądana kwota to 0,1 ethera, co oznacza 100 000 000 000 000 000 wei (jedyńka i 17 zer).



Z powodu ograniczeń JavaScriptu Remix nie potrafi przetwarzać liczb tak dużych jak 10^{17} . Dlatego należy umieścić tę wartość w cudzysłowie, aby Remix pobrał ją jako łańcuch znaków i przetwarzał jako wartość typu `BigInt`. Jeśli nie umieszysz tej wartości w cudzysłowie, środowisko IDE Remix nie przetworzy jej i wyświetli komunikat o błędzie kodowania argumentów „Error encoding arguments: Error: Assertion failed”.

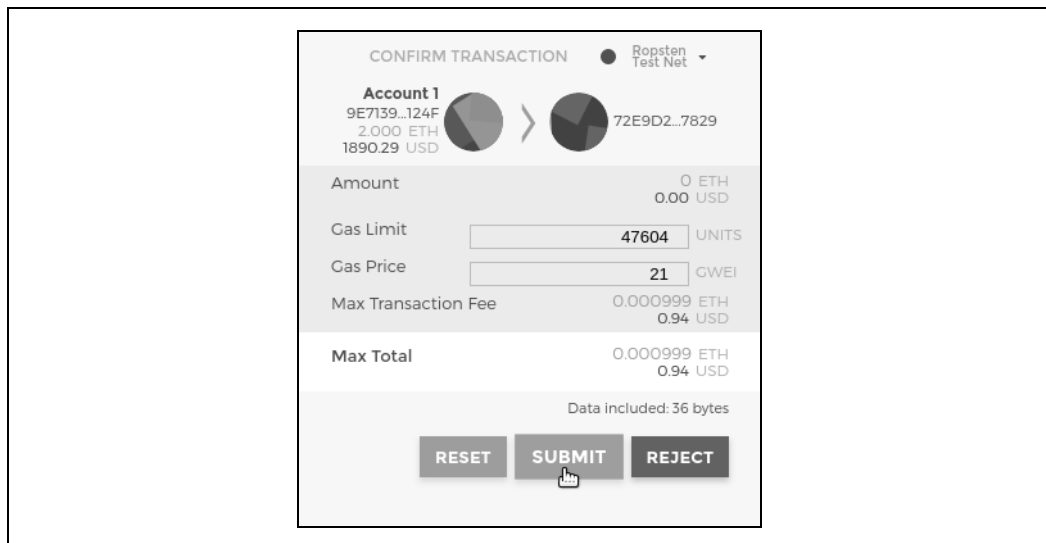
Wpisz "100000000000000000" (z cudzysłowem) w polu `withdraw_amount` i kliknij przycisk `withdraw` (rysunek 2.21).



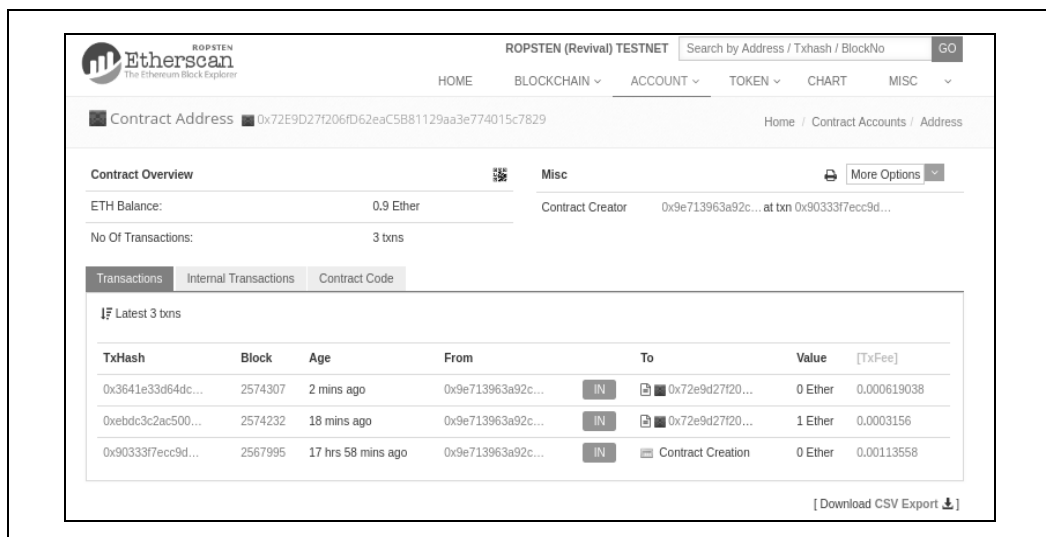
Rysunek 2.21. Kliknij przycisk `withdraw` w środowisku Remix, aby utworzyć transakcję wycofywania środków

MetaMask wyświetli okno z transakcją do zatwierdzenia. Kliknij przycisk `Submit`, aby przesłać do kontraktu żądanie wycofania środków (rysunek 2.22).

Odczekaj chwilę, a następnie odśwież stronę eksploratora bloków Etherscan, aby wyświetlić transakcję widoczną w historii adresu kontraktu `Faucet` (rysunek 2.23).



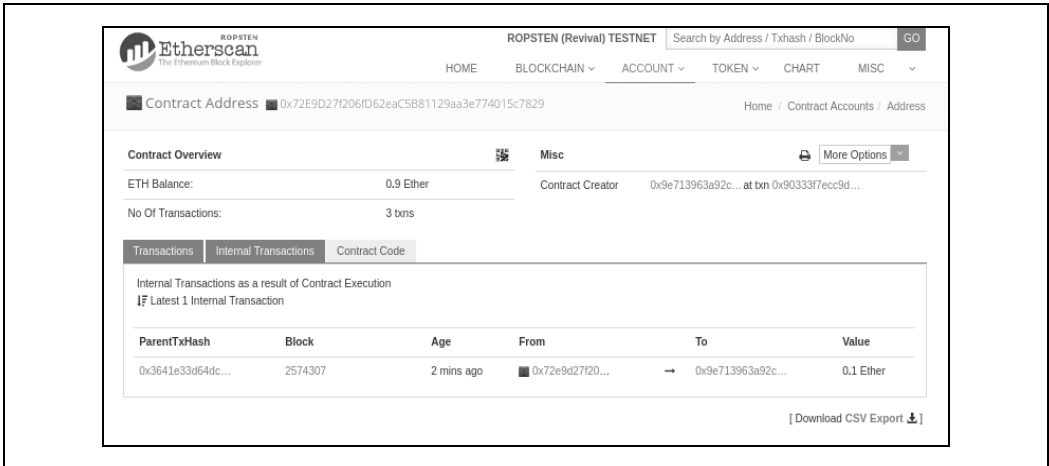
Rysunek 2.22. Transakcja w portfelu MetaMask wywołująca funkcję withdraw



Rysunek 2.23. Eksplorator Etherscan wyświetla transakcję wywołującą funkcję withdraw

Teraz widoczna jest nowa transakcja z adresem kontraktu jako adresem docelowym i wartością 0 etherów. Stan konta kontraktu się zmienił i wynosi obecnie 0,9 ethera, ponieważ zgodnie z żądaniem kontrakt przesłał 0,1 ethera. W *historii adresu kontraktu* nie widać jednak transakcji wyjściowej.

Gdzie jest transakcja wycofania środków? Na stronie historii adresu kontraktu pojawiła się nowa zakładka — *Internal Transactions*. Ponieważ transfer 0,1 ethera jest inicjowany przez kod kontraktu, wykonywana jest transakcja wewnętrzna (nazywana też **komunikatem**). Kliknij zakładkę *Internal Transactions*, aby zobaczyć jej zawartość (rysunek 2.24).



Rysunek 2.24. Eksplorator Etherscan wyświetla wewnętrzną transakcję przekazującą ethery z kontraktu

Wewnętrzna transakcja została przesłana przez kontrakt w przedstawionym wierszu kodu (z funkcji `withdraw` z kontraktu *Faucet.sol*):

```
msg.sender.transfer(withdraw_amount);
```

Oto podsumowanie: przesłałeś z portfela *MetaMask* transakcję, która zawierała instrukcje wywołania funkcji `withdraw` z argumentem `withdraw_amount` o wartości 0,1 ethera. Ta transakcja spowodowała uruchomienie kontraktu w maszynie EVM. Gdy maszyna EVM wykonywała funkcję `withdraw` kontraktu *Faucet*, najpierw wywołała funkcję `require` i upewniła się, że żądana kwota jest nie większa od maksymalnego dozwolonego poziomu 0,1 ethera. Następnie wywołana została funkcja `transfer`, aby przekazać ethery. Uruchomienie funkcji `transfer` wygenerowało wewnętrzną transakcję, która przesłała z konta kontraktu 0,1 ethera na adres portfela. Ta transakcja jest widoczna w zakładce *Internal Transactions* eksploratora Etherscan.

Podsumowanie

W tym rozdziale skonfigurowałeś portfel za pomocą rozszerzenia *MetaMask* i zasililiś go, używając kranu z sieci testowej *Ropsten*. Otrzymałeś ether na adres Ethereum Twojego portfela, a następnie odesłałeś ether na adres Ethereum kranu.

Później napisałeś w języku *Solidity* kontrakt działający jak kran. Do skompilowania kontraktu do kodu bajtowego maszyny EVM użyłeś środowiska IDE *Remix*. Następnie za pomocą tego środowiska przygotowałeś transakcję i dodałeś kontrakt *Faucet* do łańcucha bloków *Ropsten*. Utworzony kontrakt *Faucet* miał adres Ethereum, na który przesłałeś środki. W ostatnim kroku utworzyłeś transakcję, aby wywołać funkcję `withdraw`, i z powodzeniem zażądałeś 0,1 ethera. Kontrakt sprawdził żądanie i odesłał Ci 0,1 ethera za pomocą wewnętrznej transakcji.

Może wydawać się, że to niewiele, ale właśnie z powodzeniem skomunikowałeś się z oprogramowaniem, które kontroluje pieniądze w zdecentralizowanym światowym komputerze.

W rozdziale 7. napiszesz dużo więcej kodu inteligentnych kontraktów, a w rozdziale 9. poznasz najlepsze praktyki i uwagi z zakresu bezpieczeństwa.

A

ABI, application binary interface, 155
adres, 18, 89
 istniejącej instancji, 173
 kontraktu, 68
 zerowy, 137
adresy Ethereum, 102
aktywa, 236
algorytm
 pracy, 327
 dowodów stawki, 328
 ECDSA, 139, 141
 generowania podpisu, 141
 Namehash, 293
 osiągnięcia konsensusu, 38, 42
antywzorce, 190
API, 83
aplikacja
 Auction, 283
 decentralizowanie, 288
 zapisywanie, 289
 DApp, 45, 280, 283, 303
 zarządzanie, 286
aplikacje zdecentralizowane, 279
architektura maszyny EVM, 306
assembler, 181
asynchroniczne operacje, 376
ataki
 DoS, 221
 krótkie adresy, 215
 na kontrakt The DAO, 194
 parametry, 215
atesty, 236
aukcje Vickrey, 295

B

back-end, 281
Bamboo, 153
Bancor, 221
baza danych, 42
bezpieczeństwo
 ekonomiczne, 42
 inteligentnych kontraktów, 189, 377
biblioteka, 368
 EtherJar, 369
 ethers.js, 369
 Nethereum, 369
 web3.js, 368, 373, 374, 376
 web3.py, 368
Bitcoin, 37
błąd związany z wielobieżnością, 332
błędy
 przepelnienia, 186
 w adresach, 105
bomba trudności wydobywania, 327

C

Casper, 327, 328
cena paliwa, 321
cyfrowy odcisk palców, 99

D

dane
 losowe, 116
 transakcji, 133
DAO, Decentralized Autonomous Organization,
 194, 331

- DApp, decentralized application, 279
- dapp.tools, 367
- decentralizowanie aplikacji Auction, 288
- definicja kontraktu, 161
- dekompiletor, 315
- dekoratory, 184
- dezasemblacja kodu bajtowego, 315
- DNS, Domain Name System, 292
- dodawanie konstruktora, 164
- dokumenty
 - EIP, 340
 - ERC, 340
- domeny najwyższego poziomu, 294
- dostęp, 236
- dowody
 - pracy, 326, 327
 - stawki, 326, 328
- dyrektywa kompilatora, 156
- dziedziczenie
 - klas, 181
 - kontraktów, 166
- dzienniki, 187

E

- ECDSA, Elliptic Curve Digital Signature Algorithm, 139
- eksplorator bloków, 57
- Embark, 362
- Emerald, 369
- emisje ICO, 262
- ENS, Ethereum Name Service, 292, 297
- ERC, Ethereum Request for Comments, 241
- ERC20, 221
 - funkcje, 242
 - implementacje tokenów, 244
 - interfejs, 242
 - problemy z tokenami, 254
 - procesy, 243
 - przesyłanie tokenów, 250
 - struktury danych, 242
 - tworzenie tokenu, 244
 - zdarzenia, 241
- ERC223, 255
- ERC721, 258
- ERC777, 256
- Ethash, 327

- ether
 - jednostki waluty, 49
- Ethereum, 37, 39
- Ethereum Classic, 331, 335
- EthereumJS, 368
- EthereumJS helpeth, 367
- EtherJar, 369
- Ethersplay, 315
- ethery nieoczekiwane, 198
- Ethstick, 230
- EVM, 151, 305, 336
 - architektura maszyny, 306
 - instrukcje maszyny, 307
 - kod bajtowy maszyny, 311
 - kody operacji, 347
 - koszt paliwa, 347
 - wtyczki, 315
 - zużycie paliwa, 347

F

- faktoryzacja liczb pierwszych, 91
- Faucet, 64
- forki, 331, 336
- format EIP-55, 105
- formaty adresów Ethereum, 102
- front running, 219
- front-end, 281
- front-endowy interfejs użytkownika, 287
- Frontier, 40
- funkcja, 159, 162, 241
 - approve, 251
 - assert, 168
 - call, 217
 - require, 168
 - revert, 168
 - skrótów Keccak-256, 101
 - throw, 168
 - transferFrom, 251
- funkcje
 - kolejność, 184
 - przeciążanie, 182
 - pułapkowe, 91
 - skrótów, 99, 101
 - standardu ERC20, 242
 - wbudowane, 161

G

Ganache, 371
generator, 94
generowanie
 danych losowych, 114
 klucza, 121
 podrzędnego, 120
 prywatnego, 92
 publicznego, 98
 kodów mnemonicznych, 113
Geth, 42
GovernMental, 223, 225

H

hard fork The DAO, 333
hasło, 117
HD, hierarchical deterministic, 110
historia
 forków, 331
 transakcji, 59
 usług ENS, 292
Homestead, 40

I

ICAP, Inter exchange Client Address Protocol, 103
ICO, 262
IDA-Evm, 315
IDE, 64
identyfikator
 kluczy, 121
 łańcucha, 143
implementacje tokenów ERC20, 244
indeksy, 121
instalowanie
 klienta Parity, 80
 kontraktu, 314, 358
 platformy Truffle, 355
 Solidity, 154
instrukcje maszyny EVM, 307
inteligentne kontrakty, 37, 61, 149, 179
interakcja z kontraktem, 67, 374
 METoken, 248
interfejs
 ABI, 155
 ERC20, 242
 front-endowy użytkownika, 287

internetowy użytkownika, 281
JSON-RPC, 83
klientów wyroczni, 273
kontraktów tokenów, 255, 256
tokenów
 rozszerzenia, 261
IPFS, Inter-Planetary File System, 282

J

Jaxx, 87
jednostki waluty ether, 49
język
 Bamboo, 153
 LLL, 152
 Serpent, 152
 Solidity, 149, 153, 273
 Vyper, 153, 179
języki w Ethereum, 151
JSON-RPC API, 83

K

Keccak-256, 101
klient, 42
 Ethereum, 73, 77
 Geth, 81
 Go-Ethereum, 80
 Parity, 79
 RPC, 56
klienty zdalne, 85
klonowanie repozytorium, 81
klucze, 89
 podrzędne, 120
 prywatne, 18, 92
 publiczne, 90, 94, 98
 rozszerzone, 119
klucz-wartość, 41
kod
 bajtowy maszyny, 311
 do instalowania kontraktu, 314
 mnemoniczny, 108, 110, 112, 117
kodowanie
 Basic, 103
 Direct, 103
 Indirect, 104
 szesnastkowe, 104
kody operacji, 347
kompilator solc, 155

- kompiłowanie kontraktu Faucet, 64
- kompletność w sensie Turinga, 43, 44, 319
- komponenty
 - Ethereum, 42
 - łańcucha bloków, 38
- komunikat, 38, 71, 100
- konfigurowanie platformy truffle, 357
- konkurencja, 329
- konsensus, 325
 - dowody pracy, 326
 - dowody stawki, 326
 - reguły, 329
- konsola platformy Truffle, 248, 360
- konstruktor kontraktu, 163
- konstruktory, 225
- konta
 - EOA, 61, 135
 - kontraktów, 61
- kontekst
 - bloku, 160
 - maszyny EVM, 150
 - transakcji, 160
- kontrakt, 135, 137, 161
 - Etherpot, 218
 - Ethstick, 230
 - Faucet, 64, 164
 - King of the Ether, 218
 - METoken, 247
 - The DAO, 194, 331
- kontrakty
 - wywoływanie, 172
 - bibliotek, 232
 - instalowanie, 314
 - inteligentne, 149, 179
 - back-endu, 284
 - bezpieczeństwo, 189, 377
 - cykl życia, 150
 - testowanie, 370
 - rozrzutne, 179
 - samobójcze, 179
 - zachłanne, 179
 - zewnętrzne, 210
- kontrola, 51
- kontrowersje, 329
- koszt paliwa, 177, 347
- kryptety, 272
- kryptografia, 89
 - klucza publicznego, 90
 - krzywej eliptycznej, 91, 95

- kryptograficzne funkcje skrótu, 99
- kryptowaluta ether, 37
- krzywa eliptyczna, 95, 99

L

- liczby zmiennoprzecinkowe, 229
- limit paliwa, 322
- LLL, 152
- localhost 8545, 56
- logarytm dyskretny, 91
- losowość, 209
- LTS, long-term support, 355
- luka, 191, 195, 198, 202, 208, 209, 210, 216, 217, 219, 221, 224, 225, 227, 229, 231

Ł

- łańcuch bloków, 38
 - dla programistów, 47
 - Ganache, 371
 - o ogólnym przeznaczeniu, 41
 - ogólnego użytku, 45
 - pierwsza synchronizacja, 82
 - rejestrwanie danych, 147
 - symulowanie pracy, 76
 - tworzenie kontraktu, 66

M

- maszyna
 - stanowa, 38, 41, 42
 - Turinga, 43
 - wirtualna, 38, 307
 - Ethereum, 305
 - EVM, 42, 151, 305
 - SputnikVM, 368
- menedżer npm, 363
- MetaMask, 86
- METoken, 248
- Metropolis, 40
- Mist, 87
- mnemoniczna kopia zapasowa, 53
- modyfikator, 114, 180
 - funkcji, 165
- MyCrypto, 87
- MyEtherWallet, 87

N

- narzędzie, 367
 - EthereumJS, 368
 - EthereumJS helpeth, 367
 - truffle, 171
- Nethereum, 369
- niedopełnienie arytmetyczne, 194
- nieodłączność, 237

O

- obiekt address, 160
- obliczanie zużycia paliwa, 320, 321
- obsługa błędów, 168
- odbiorca transakcji, 132
- odpowiedzialność, 51
- odsyłacze, 377
- odzyskiwanie klucza publicznego, 144
- OpenZeppelin, 363
- operacja
 - CALL, 202
 - DELEGATECALL, 202
- operacje
 - arytmetyczne, 97
 - właściciela, 222
- osiąganie konsensusu, 38, 325, 326, 329

P

- pakiet dapp.tools, 367
- paliwo, 45, 176, 319, 320
 - cena, 321
 - limit, 322
 - obliczanie zużycia, 320, 321
 - ujemne zużycie, 322
- para kluczy, 91
- Parity, 42
- pełny węzeł, 75
- piramida finansowa, 223, 225, 226
- platforma
 - Embark, 362
 - Emerald, 369
 - OpenZeppelin, 363
 - truffle, 357, 358
 - Truffle, 355, 356, 360
 - ZeppelinOS, 366
- płatność, 133, 135
- poddomeny, 301

- podpisy cyfrowe, 139, 140
- podpisywanie
 - transakcji, 142, 143, 146
 - w trybie offline, 145
- polecenie
 - delegatecall, 173
 - selfdestruct, 164
 - SELFDESTRUCT, 163
- Porosity, 315
- portfel, 50, 107
 - HD, 110, 118
 - MetaMask, 52
 - Parity, 134
 - Parity z wielopodpisem, 206, 208
- portfele
 - deterministyczne, 108, 110
 - działające w przeglądarkach, 86
 - mobilne, 86
 - niedeterministyczne, 108
 - struktura drzewiasta, 122
 - zalecane praktyki, 112
- PoW, proof of work, 326
- PoWHC, Proof of Weak Hands Coin, 198
- poziomy widoczności, 207
- praktyki dotyczące portfeli, 112
- prawo głosu, 236
- precyzja, 229
- problem
 - logarytmu dyskretnego, 91
 - stopu, 43, 319
- procesy, 243
- program helpeth, 367
- programowanie defensywne, 189
- programy komputerowe, 149
- protokoły przekazywania komunikatów, 282
- protokół ICAP, 103
- prototyp, 136
- przechowywanie danych, 282
- przechwytywanie zdarzeń, 170
- przeciążanie funkcji, 182
- przeciwwobraz, 100
- przedmioty kolekcjonerskie, 236
- przekazywanie
 - danych, 135
 - środków, 135
- przepełnienie, 186, 194
- przesyłanie
 - plików, 290
 - tokenów ERC20, 250

przynęta, 214
punkt generatora, 94
puste dane wejściowe, 101

R

reguły osiągnięcia konsensusu, 42
rejestrowanie
 danych, 147
 nazwy, 297
Remix, 64
repozytorium, 81
resolwery, 295, 301
rozsyłanie transakcji, 147
rozwój Ethereum, 40, 46
RPC, Remote Procedure Call, 83
Rubixi, 226
rzutowanie typów zmiennych, 182

S

Serenity, 40
Serpent, 152
sieci publiczne testowe, 75
sieć
 Ethereum, 55, 74
 P2P, 38, 42
 testowa
 Kovan, 55
 Rinkeby, 56
 Ropsten, 55
skierowany graf acykliczny, 327
skrypt, 373
skrypty instalacyjne, 358
słowo kluczowe new, 172
Solidity, 149, 153
 dodawanie konstruktora, 164
 dziedziczenie kontraktów, 166
 funkcje, 159, 162
 instalowanie, 154
 interfejsy klientów wyroczni, 273
 kompilator, 155, 156
 kompilowanie kodu, 311
 konstruktor kontraktu, 163
 kontrakt, 161
 modyfikatory funkcji, 165
 obsługa błędów, 168
 programowanie, 157
 typy danych, 157

 wersje języka, 153, 156
 zdarzenia, 169
 zmiennie globalne, 159
specyfikacja usługi ENS, 292
sprawdzanie
 niezmienników, 198
 poprawności podpisu, 140
SputnikVM, 368
stan w Ethereum, 310
standard, 339
 EIP-155, 143
 ERC20, 241
 ERC223, 255
 ERC721, 258
 ERC777, 256
standardy dotyczące tokenów, 260
struktura
 drzewiasta portfela, 122
 transakcji, 125
struktury danych, 42, 242
Swarm, 46, 282, 289
system
 nagród, 38
 Swarm, 289
 przesyłanie plików, 290
 tłumaczenie nazwy na skrót, 302
szacowanie kosztów paliwa, 177

Ś

ścieżka, 119, 121
śledzenie wartości nonce, 127
środowisko
 Node.js, 373
 programistyczne, 154

T

tablice dynamiczne, 177
techniki zapobiegania, 193, 197, 201, 206, 210, 213,
 217, 218, 220, 223, 225, 228, 230, 232
testowanie
 inteligentnych kontraktów, 370
 w łańcuchu bloków, 371
testowy ether, 61
token użytkowy, 236
tokeny, 235, 237, 378
 deed, 296
 ERC20, 244

- rozszerzenia standardów, 261
- stosowanie standardów, 260
- narzędziowe, 239
- w Ethereum, 240
- tożsamość, 236
- transakcje, 18, 42, 125
 - atomowe, 151, 168
 - dane, 133
 - nieprzetworzone, 143
 - odbiorca, 132
 - podpisywanie, 142, 143, 146
 - rozsyłanie, 147
 - ryzyko, 237
 - specjalne, 137
 - struktura, 125
 - wartość, 133
 - wartość nonce, 126
 - z wieloma podpisami, 148
- transfer zbiorczy, 198
- transferFrom, 243
- Truffle, 355
 - Box, 356
 - instalowanie kontraktów, 358
 - instalowanie platformy, 355
 - konfigurowanie platformy, 357
 - konsola platformy, 360
 - migracje, 358
 - tworzenie katalogu projektu, 356
- tryb
 - asynchroniczny, 373
 - zgodności z Geth, 85
- Turing, 43
- tworzenie
 - kontraktu, 66, 137, 172
 - nieprzetworzonych transakcji, 143
 - podpisu cyfrowego, 140
 - portfela, 53
 - portfela HD, 118
 - tokenu ERC20, 244
- typy danych, 157

U

- udziały, 236
- uniwersalna
 - maszyna Turinga, 43
 - obliczalność, 43
- uruchamianie klienta, 83
 - Ethereum, 77

- usługa
 - ENS, 292, 297, 299
 - resolwery, 301
 - tworzenie poddomeny, 301
 - zarządzanie nazwą, 300
 - Ethereum Name Service, 292
- uwierzytelnianie, 230
 - danych, 270
- używanie
 - wyroczeni, 266
 - tokenów, 238

V

- Vyper, 153, 179
 - błędy przepełnienia, 186
 - dekoratory, 184
 - dziedziczenie klas, 181
 - funkcje, 184
 - kompilator, 185
 - modyfikatory, 180
 - odczyt danych, 186
 - przeciążanie funkcji, 182
 - rzutowanie typów, 182
 - warunki końcowe, 183
 - warunki wstępne, 183
 - zapis danych, 186
 - zmiany stanu, 183
 - zmiennne, 184

W

- waluta, 236
 - narzędziowa, 37
 - PoWHC, 198
- wartość
 - nonce, 126, 130
 - luki, 129
 - śledzenie, 127
 - zatwierdzanie, 129
 - transakcji, 133
- web3.js, 368, 373–376
- web3.py, 368
- wektor testowy, 101
- wersje języka Solidity, 153
- wewnętrzność, 237
- węzeł, 74
- węzły .eth, 295
- Whisper, 46, 282

- widoczność, 207
- wielobieżność, 190, 214, 332
- wielopodpis, 206, 208
- własność węzła korzenia, 294
- wskaźniki do pamięci trwałej, 226
- współbieżność, 130
- wybieranie portfela, 50
- wycofywanie środków, 69
- wykrywanie błędów, 105
- wymagania
 - programowe, 78
 - sprzętowe, 77
- wymiennosc, 237
- wyroczenie, 265
 - interfejsy klientów, 273
 - obliczeniowe, 271
 - przypadki użycia, 266
 - wzorce projektowe, 267
 - zdecentralizowane, 272
- wysyłanie etherów, 58
- wyścig, 219
- wyświetlanie adresu kontraktu, 68
- wywołania, 133
 - nieprzetworzone, 173
 - wsadowe, 84
- wywołanie await, 376
- wywoływanie
 - komunikatów, 159
 - kontraktów, 172, 177

- wzorce
 - projektowe wyroczeni, 267
 - wypłacania, 218
- wzorcowy specyfikacja, 38

Z

- zapobieganie, *Patrz* techniki zapobiegania
- zarządzanie aplikacją DApp, 286
- zasilanie kontraktu, 68
- zasób, 236
- zdarzenia, 169, 241
 - przechwytywanie, 170
- zdecentralizowany komputer światowy, 150
- zdobywanie testowych etherów, 56
- ZeppelinOS, 366
- ziarna, 108, 110, 113
- złudzenie losowości, 209
- zmiana stanu, 41
- zmienianie sieci, 55
- zmiennic globalne, 159
- znacznik czasu bloku, 224
- zużycie paliwa, 320, 321, 347

Ź

- źródło transakcji, 130

PROGRAM PARTNERSKI

— GRUPY HELION —

1. ZAREJESTRUJ SIĘ
2. PREZENTUJ KSIĄŻKI
3. ZBIERAJ PROWIZJĘ

Zmień swoją stronę WWW w działający bankomat!

Dowiedz się więcej i dołącz już dzisiaj!

<http://program-partnerski.helion.pl>

GRUPA
Helion

Nowy wymiar innowacji w programowaniu!

Ethereum to platforma służąca do obsługi kryptowalut. Została zaprojektowana z zachowaniem wysokich standardów bezpieczeństwa i transparentności. Umożliwia uruchamianie zdecentralizowanych aplikacji (DApp) i inteligentnych kontraktów, które nie mają scentralizowanej kontroli ani pojedynczego punktu podatności na awarie i są zintegrowane z siecią obsługi płatności oraz operują na otwartym łańcuchu bloków. Ethereum cieszy się dużym zainteresowaniem takich organizacji jak IBM, Microsoft, NASDAQ, które coraz bardziej angażują się w korzystanie z tej platformy.

Ta książka jest praktycznym poradnikiem i encyklopedycznym źródłem wiedzy o Ethereum przeznaczonym dla programistów, którzy chcą przyswoić wiadomości dotyczące łańcuchów bloków oraz tworzenia inteligentnych kontraktów i zdecentralizowanych aplikacji – DApp. Znalazły się tu zarówno podstawowe informacje, jak i szereg zaawansowanych zagadnień związanych z Ethereum. Opisano, w jaki sposób tworzy się w nim transakcje, przedstawiono kwestie związane z kluczem publicznym, skrótami i podpisami cyfrowymi. Poszczególne zagadnienia, a także najlepsze praktyki, wzorce projektowe i antywzorce z obszaru bezpieczeństwa uzupełniono starannie opracowanymi przykładami kodu.

W tej książce między innymi:


- uruchamianie klienta Ethereum
- korzystanie z portfeli cyfrowych
- interakcje z klientami Ethereum za pomocą wywołań RPC
- tokeny reprezentujące aktywa, udziały czy głosy
- budowa zdecentralizowanych aplikacji z użyciem komponentów w modelu P2P

Andreas M. Antonopoulos

jest uznanym autorytetem w dziedzinie kryptowalut i łańcuchów bloków. Jest też znany z wystąpień, w których łączy ekonomię, psychologię i teorię gier z technologią i historią. W ten sposób sprawia, że najbardziej abstrakcyjne zagadnienia stają się zrozumiałe i rzeczywiste.

Dr Gavin Wood

jest programistą, technologiem, wykładowcą i projektantem gier. Jest też architektem Ethereum. Zaprojektował język kontraktów Solidity i napisał specyfikację typu Yellow Paper – pierwszą formalną specyfikację protokołu łańcucha bloków.

Helion helion.pl

HELION SA
ul. Kościuszki 1c
44-100 Gliwice
tel.: 32 230 98 63
helion@helion.pl

Sprawdź nasze szkolenia

SZKOLENIA



AKADEMIA IT & BUSINESS

WWW.SZKOLENIA.HELION.PLKOD KORZYŚCI
Sięgnij po więcej! ▶

ISBN 978-83-283-5574-3



9 788328 355743