cryptography

MDPI

Article

Cybersecurity Test Bed for Smart Contracts

Casimer DeCusatis *, Brian Gormanly, John Iacino, Reed Percelay, Alex Pingue and Justin Valdez

School of Computer Science and Mathematics, Marist College, Poughkeepsie, NY 12601, USA
* Correspondence: casimer.decusatis@marist.edu; Tel.: +1-845-575-3883

Abstract: Blockchain, smart contracts, and related concepts have emerged in recent years as a promising technology for cryptocurrency, NFTs, and other areas. However, there are still many security issues that must be addressed as these technologies evolve. This paper reviews some of the leading social engineering attacks on smart contracts, as well as several vulnerabilities which result from insecure code development. A smart contract test bed is constructed using Solidity and a Metamask wallet to evaluate vulnerabilities such as insecure arithmetic, denial of service, and re-entrancy attacks. Cross-chain vulnerabilities and potential vulnerabilities resulting from layer 2 side-chain processing were also investigated. Mitigation best practices are proposed based on the experimental results.

Keywords: blockchain; smart contract; cybersecurity

Citation: DeCusatis, C.; Gormanly, B.; Iacino, J.; Percelay, R.; Pingue, A.; Valdez, J. Cybersecurity Test Bed for Smart Contracts. *Cryptography* **2023**, 7, 15. https://doi.org/10.3390/cryptography7010015

Academic Editor: Kentaroh Toyoda

Received: 14 February 2023
Revised: 2 March 2023
Accepted: 3 March 2023
Published: 10 March 2023

1. Introduction

Blockchain provides an immutable database that operates as a distributed transaction ledger. It has been used for cryptocurrencies and so-called Web3 technologies [1,2]. Some blockchain frameworks, such as Etherium and the Etherium Virtual Machine, are also well suited for the development of smart contracts, which automate transactions in accordance with pre-defined specifications and use digital signatures to approve each step of the contract. Smart contracts have been proposed to supplement or replace existing contractual methods in a wide range of applications [2]. This includes trading of digital currencies and non-fungible tokens (NFTs), which have received a great deal of attention recently and in some cases may have significant issues [3]. While smart contracts can be quite complex, the following brief overview will contextualize this research.

Smart contracts are commonly coded in the language Solidity (other alternatives include Vyper, Rust, and Javascript). Solidity is a statistically typed, object-oriented language which sets itself apart through special state variables that allow for access control to be written into the program. This unique addition enhances the security potential of any contract, if used properly. Solidity uses a syntax similar to ECMAScript which makes it easier for existing web developers to deploy smart contracts. ECMAScript is a JavaScript standard intended originally to ensure web pages interoperate correctly when different browsers are used. It is commonly used for client-side scripting, and it is increasingly being used for writing server-side applications and services using runtime environments such as Node.js. ECMAScript has been formalized through the use of operational semantics, a category of formal programming language semantics in which certain desirable properties of a smart contract are verified by constructing proofs from logical statements about the contract's execution and procedures. This stands in contrast to denotational semantics, which attaches mathematical meaning to the terms of a smart contract to validate its desired properties. Operational semantics can be used to validate a contract's security, safety, or correctness, for example. Solidity benefits from these techniques, which both make it accessible to web developers and provide the means to validate security of the resulting contract. However, unlike ECMAScript, Solidity employs static variable typing and variadic return types. Solidity also differs from other Ethereum virtual machine supported languages in several

important respects. Solidity supports complex member variables for contracts, including arbitrary hierarchical mappings and constructs. Smart contracts developed in Solidity support inheritance, including multiple inheritance with superclass linearization (i.e., a mechanism used in object-oriented languages to resolve conflicts when inheriting different definitions of the same property from multiple superclasses). Further, Solidity introduces an application binary interface that facilitates many different type-safe functions within a single contract; this property includes natural language specification (a documentation system for specifying user-centric descriptions of the ramifications of method calls). When properly deployed, these features can help mitigate security vulnerabilities in a smart contract; however, they do not guarantee that such a contract is immune from the attacks discussed later in this paper.

To increase the scalability of blockchains such as Ethereum, layer-2 smart contract solutions were developed. These solutions enable off-chain processing and grouping of transactions into similar batches, thereby reducing the overall cost of adding information to the blockchain as well as increasing the overall number of transactions that can be handled by the blockchain. This approach can potentially lower the Etherium transaction cost (measured in gas). Examples include platforms such as Polygon, Arbitrum, and Optimism, which approach this problem in different ways. Polygon employs zero-knowledge rollups, which combine a large number of transactions that were executed off-chain and submit them as one transaction to Etherium [4]. The approach is based on zero knowledge proofs, which refers to a way of proving that you know something without revealing what it is that you know. Zero-knowledge rollups batch together many layer-2 transactions off chain into one transaction on the Etherium mainnet. This single transaction constitutes a validity proof, since the mainnet is only updated once this transaction has been verified. These batches may contain thousands of transactions. The ability to verify a large number of transactions at once improves scalability and performance. Using this approach, Etherium's raw processing rate of about 15 to 30 transactions per second can be increased to over 40,000 transactions per second. By comparison, Arbitrum uses an optimistic rollup, which posts transactions as soon as they are validated and allows anyone to dispute the result during an interactive challenge period [5]. Suspicious transactions are processed off-chain by sending part of the transaction back through the Etherium virtual machine. Smart contracts are vulnerable to a wide range of cybersecurity attacks, including malicious code, social engineering, and more. Further, the addition of outside technologies such as off-chain layer 2 processing or cross-chain bridging increases the attack surface. In 2022 nearly USD 2 billion was stolen due to vulnerabilities in cross-chain bridges alone [6].

In this paper, a smart contract test bed was built to evaluate several of these vulnerabilities, and best practices for attack mitigation recommended based on the experimental results. This work experimentally demonstrates attacks and mitigation using insecure arithmetic, denial of service, and re-entrancy. Related approaches including a variety of social engineering attacks and cross-chain bridging attacks and their mitigation are also discussed.

This approach has not previously been discussed in the literature. Very recent systematic reviews of the technical literature [7] focus on different issues such as plagiarism and copyright infringement. Formal methods of blockchain smart contract verification have been discussed [8] but without attention to the software-based security vulnerabilities in this work. There is some discussion in the literature of vulnerability detection in contract validation [9]; however, this does not address experimental validation of the issues noted in the test bed. The literature discusses formal verification, or how to evaluate if a contract behaves in the desired manner. Expected behavior of the contract can be described using formal modeling, and specification languages can be used to create formal properties. A formal model is an abstract mathematical description of the contract, which are used to establish if the contract is functionally correct. There has been a great deal of work on different approaches to smart contract validation [10–12]; however, this body of work does not categorize security vulnerabilities which can be exploited to subvert an otherwise functionally correct contract. There have been related studies of blockchain management

systems [13–15] which are more concerned with how to manage and maintain smart contract functionality rather than how to mitigate security vulnerabilities. Further, none of this prior art has formally classified the major vulnerabilities according to the industry standardized Common Vulnerability Scoring System (CVSS) [16]. Thus, contributions from this research investigate areas which have not previously received as much attention from blockchain researchers.

The remainder of this paper is organized as follows. Following the introduction, the design of the test bed is described. Next, there is a discussion of common social engineering attacks which have been demonstrated against smart contracts. Then, there is an investigation of code-based attacks, including insecure arithmetic, denial of service, and re-entrancy attacks. Cross-chain bridging attacks are also briefly discussed. Experimental results are then summarized along with recommended best practices for attack mitigations.

2. Materials and Methods for the Smart Contract Test Bed

The smart contract test bed is shown in Figure 1. A locally hosted website for an Etherium node was created for this testing, as well as a Metamask wallet. Both the Etherium node and Metamask wallet code are available open source and were customized for use in this test bed. This arrangement can be used to test the effect of vulnerabilities in smart contract code and digital wallet code, and to propose mitigations. This design also allows the test bed to bridge transactions between Etherium and Arbitrum, which makes it suitable for testing cross-chain bridge attacks (as discussed later in this paper). Draft smart contracts are compiled into Javascript bytecode. A Metamask digital wallet is connected to our host website using Javascript code; the compiled smart contract results are inserted into the wallet using Javascript as well. In this way, the smart contract interacts with the digital wallet, host website, and any required web3 libraries (several library options are discussed later in this paper). The Web3 libraries are incorporated into our host website using HTML code in the website. This environment used widely available Intel x86 hardware with 1 Gbit/s network interface cards and is typical of those used for cryptocurrency or NFT trading, as well as other smart contract applications. The test bed consists of two servers occupying two data center rack units.

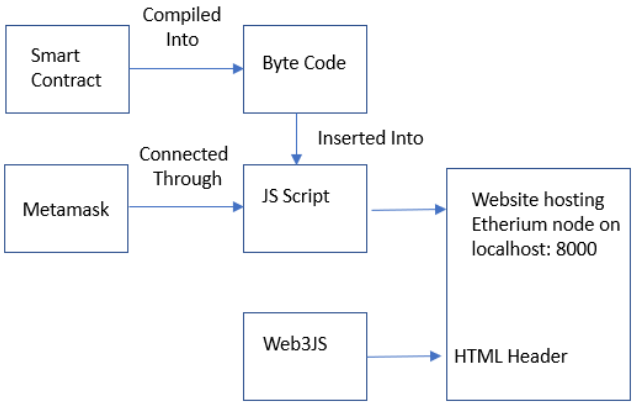

Figure 1. Smart contract test bed.

The Remix independent development environment (IDE) was used to write, debug, and compile smart contracts. This is available online from remix.etherium.org, as the Remix Desktop client, or as an extension to VSCode. This environment was chosen because it is representative of many commercially available smart contracts. The IDE requires minimal setup, fosters a rapid development cycle, and features a wide variety of plugins with intuitive graphical user interfaces. Further, this environment reportedly offers a faster execution time than alternatives in serving both static and dynamic content. A sample screenshot is provided in Figure 2. On the far left is a vertical icon menu, which is used to select plugins that will appear in the adjacent panels (in Figure 2, this is annotated with a

red circle around the plugin currently in use). Remix allows the system to only load the functionality required; other plugins can be turned off. Since most functionality in Remix is implemented through plugins, the plugin manager is an important part of this screen. The file explorer window on the left is used for navigation and also allows the developer to view/set parameters for smart contract deployment, such as viewing calculated gas fees. For example, the button highlighted in red is used to choose a test environment and gas fees. The window on the right shows the actual smart contract code, and allows execution of application code, displays debugging and error messages, and may be used to execute other scripts as well. This window can also be used to compile files or plugins from the independent development environment. Additional workspaces can be opened to help separate and organize projects. It is possible to associate workspaces with a Git, or to clone a repo using the appropriate plugin. Code written in Remix is saved to the browser storage by default, which is not permanent; inadvertently clearing the browser storage cache will permanently delete all files stored there, unless they are backed up to another location.

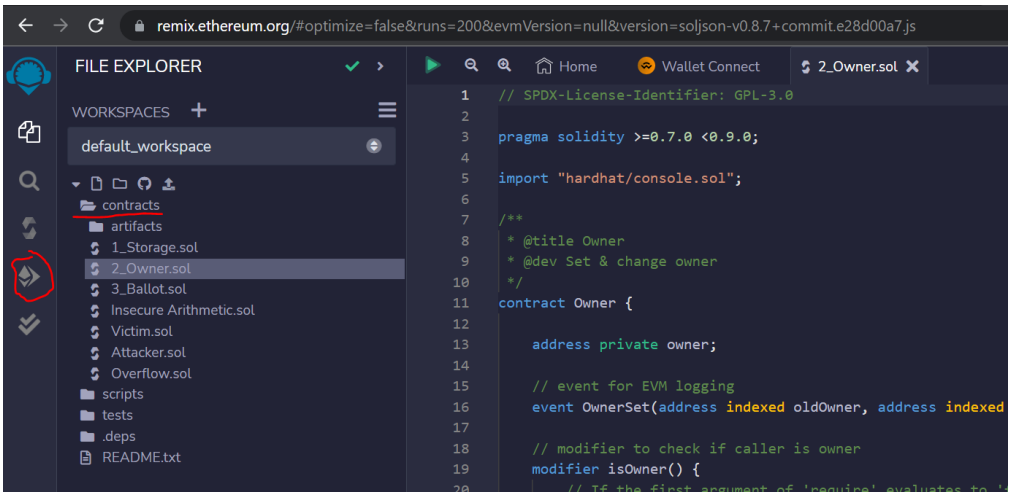

Figure 2. Sample screen shot of smart contract IDE.

3. Results and Discussion for Social Engineering Attacks

Due to the currently unregulated nature of most smart contracts, there are a wide range of social engineering attacks which do not require a high level of technical expertise. Most of these assume that trading in cryptocurrencies or NFTs is a zero-sum game [17]. These may be combined with code-based attacks, as will be discussed in a later section. Unless system administrators and others with access to sensitive information (such as private keys and verification nodes) are regularly trained to resist social engineering, even a well-crafted and tested smart contract may be compromised. Some of the major types of social engineering attacks that have been observed in online marketplaces are reviewed in the following sections.

3.1. Pump and Dump Attacks

"Pump and Dump" attacks are not a new invention, although they are prohibited by more conventional financial regulations. Digital currencies have proven to be an excellent opportunity for their implementation with few repercussions. Pump and dumps typically consist of an attacker picking an existing youthful asset (such as a struggling cryptocoin) and purchasing a large amount at a low price. This generates false "hype" and encourages others to invest in the asset, artificially driving up the price. Social media forums are used to disseminate false information and generate more hype around the asset. Potential investors are encouraged to act quickly to avoid missing out on profits. The attacker waits for the value of the asset to inflate, and then sells their majority holdings to cash out and make a profit. This typically results in collapse of the asset value, and other investors are

left with near valueless coins. One of the largest such attacks on record resulted in the loss of over USD 1 billion [18].

3.2. Rug Pull

Rug pulls are very similar in overall design to pump and dump schemes. The main difference is that rug pulls are conducted by more highly skilled attackers who develop a new asset entirely with the intent of cashing out and causing the asset value to collapse. The attackers are also the asset developers. Since they are present at the creation of the asset, they not only have the largest profit potential but also can claim to have a personal stake in the asset success, giving investors a reason to trust them while they hype the value of the asset. Since the developers often take a more public role in promoting the asset, they typically need to become anonymous once the fraud is uncovered and the asset value collapses. The victim often has little recourse, since they may not know the real name or location of the attacker. In one example over USD 14 M worth of cryptocurrency was stolen [19].

3.3. Wash Trading

Wash trading is an attack that consists of several parties cooperatively trading an asset with each other, in order to manipulate the market value. This creates the perception of increased trading volume, which artificially raises the asset value, or reduced trading to devalue an asset (purposely making it unstable or making it a candidate for a bulk investment at a lower price). The collusion of parties in the crypto market is harmful because assets can be devalued, invested in, artificially appreciated, and then dumped which can destabilize the market and affect the value of assets held by genuine investors. This scam requires less outside investment than the pump and dump and can easily be perpetrated again and again by cooperating parties on a variety of different trading assets. Recent research suggests that over USD 30 B of NFT trading volume on Etherium is actually wash trading [20].

3.4. Red Queen's Race

Red Queen's Race is named after a passage from Lewis Carroll's *Through the Looking-Glass*, where Alice runs as fast as she can but is only able to stay in the same place; the Red Queen tells her that she must run twice as fast to move herself. This attack refers to the rapid fluctuation in asset value in crypto markets, which can quickly make investments less profitable than intended. For example, if malicious actors execute a pump and dump scheme at nearly the same time as a genuine investor obtains their stake, the legitimate investor may lose their expected profit as the asset rapidly depreciates; they are stuck in the same place they started, or may be even worse off. The effect has been described in the context of security in an article published by the International Systems Security Agency [21] and in the context of blockchain-based transactions [22].

3.5. Gold Brick

Gold brick attacks entail deceivingly selling an asset for more than it is actually worth. The name is derived from historical scams of selling ingots that are simply plated with gold, as opposed to being solid gold, for the price of a solid gold ingot. In the same way that it is hard to tell if the ingot is indeed solid gold, it can sometimes be hard to tell what assets are indeed worth their marketed price. NFTs are particularly susceptible to this effect, since it is difficult to assign objective worth to the asset; it is only worth what someone else is willing to pay at any given time. This technique may be combined with the social engineering attacks discussed previously.

3.6. Flash Loans

Flash loans are uncollateralized loans that must be paid back in one blockchain transaction or else the loan is reversed. Their name reflects that such loans occur very quickly, and

the hasty turnaround is aided by decentralized smart contracts. This technology provides yet another opportunity to commit fraud, as large amounts of flash loans can be taken out simultaneously leading to temporary manipulation of an asset's market value until the loans are reversed. Flash loan scams offer a relatively low risk market manipulation attack. Two recent flash loan attacks resulted in the loss of nearly USD 1 M [23].

4. Results and Discussion for Code-Based Attacks

A smart contract is fundamentally just code, and thus, is subject to attacks based on vulnerabilities inherent in that code. These practices include cryptojacking (user devices are infiltrated with malware, and processing power is stolen to gain mining rewards) and griefing (deliberately acting to ensure transactions fail, in order to cause chaos for legitimate users; the name is taken from the video game community where it describes individuals who act in bad faith). In the following sections, different code-based attacks are investigated. The vulnerability is first described, and a hypothesis for an attack vector is formed. The test bed from Figure 1 is used to demonstrate an effective attack, and to test whether a proposed attack mitigation is successful.

4.1. Insecure Arithmetic Attack

Smart contracts for financial transactions need to conduct checks to prevent overdrafts (if a wallet balance is low enough) or to prevent overflows (if a wallet balance is too large). Insecure arithmetic refers to the use of arithmetic operations in a Solidity smart contract that can lead to unexpected or incorrect results. Specifically, this refers to certain conditions not being properly checked before an operation is performed, and most commonly results in operations dividing by zero or operations resulting in overflow/underflow of integer values.

In Solidity, integer values have a fixed size and can therefore only hold a limited range of values. For example, an uint8 (unsigned 8-bit integer) can hold values from 0 to 255. If a value is assigned to an uint8 that is not in this range, it can be hypothesized that this will cause an overflow or underflow, resulting in an incorrect value being stored in the variable. When the value is too large, an overflow occurs, and if the value is too low, an underflow occurs. Likewise, an uint16 or uint256 should not be assigned values outside its specified range. The feasibility of using this approach to induce an overflow condition is illustrated by sample code from the smart contract test bed shown in Figure 3, in which an uint256 is incorrectly employed in balance/overflow checking code.

```
function transfer(address _to, uint256 _value) {
    /* Check if sender has balance and for overflows */
    require(balanceOf[msg.sender] >= _value && balanceOf[_to] + _value >= balanceOf[_to]);

    /* Add and subtract new balances */
    balanceOf[msg.sender] -= _value;
    balanceOf[_to] += _value;
}
```

Figure 3. Example of insecure arithmetic code.

Dividing by zero is another example of potential risk for insecure arithmetic in Solidity. This is because Solidity, unlike most programming languages, does not throw an error when the code attempts to divide by zero. Instead, Solidity just returns a value of zero. This incorrect operation can lead to unexpected results which an attacker can exploit. A trivial code error in the smart contracts used by our test bed demonstrates that a divide by zero error goes undetected under normal operating conditions.

To mitigate these risks, there are a few best practices that are recommended for developers. First, use the appropriate data types for variables. In Solidity, different data types have varying ranges of values that they can hold. It is vital to use the appropriate data type for each variable to avoid overflow/underflow conditions. Secondly, use the

SafeMath library, a commonly used library in Solidity that provides a set of mathematical functions which automatically check for overflow and underflow conditions and revert the transaction if they occur (SafeMath is recommended by smart contract auditing firms such as OpenZeppelin [24]). Lastly, test and audit the contract. Before deploying a contract to the blockchain, it is important to design and execute a thorough test plant to identify any potential vulnerabilities, including those related to insecure arithmetic. Once the smart contract is on the blockchain, it is immutable, and any vulnerabilities can and will be exploited. To test these approaches, a smart contract instance was created in the test bed, which induced an overflow condition as shown in Figure 3. This condition was detected using a test plan which analyzes variable types against the range of potential values they are designed to hold. By changing the variable type, we experimentally demonstrated that the error condition was corrected. A second new contract was created with the same error, which was again successfully mitigated by deploying the SafeMath library in the test bed. A third new contract was then created with a deliberate divide by zero error; this was identified and mitigated by deploying the SafeMath library. We repeated this process five times for each of the three insecure arithmetic errors to validate that these recommendations worked consistently under a variety of input conditions.

Results have been analyzed using the Common Vulnerability Scoring System (CVSS) version 3.1 [16], a method used by the National Institute of Standards and Technology (NIST) to measure vulnerability severity. CVSS consists of three metric groups: base, temporal, and environmental. The base metrics produce a score ranging from 0 to 10, which can then be modified by scoring the temporal and environmental metrics. A CVSS score is also represented as a vector string, a compressed textual representation of the values used to derive the score. The CVSS is well established as a standard measurement system for consistent vulnerability severity scores, which may be used as a factor in prioritizing vulnerability remediation activities. Results of the CVSS analysis for insecure arithmetic attacks are shown in Figure 4, along with the corresponding vector string, following the standard approach described in [16].

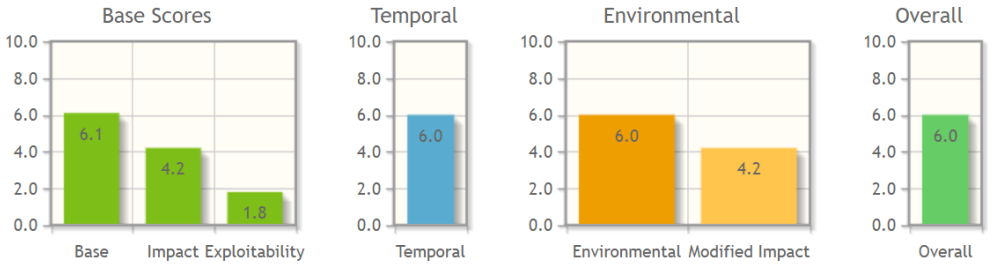

Figure 4. CVSS scores for insecure arithmetic; the corresponding vector string is AV:L/AC:L/PR:L/UI:N/S:U/C:N/I:H/A:L/E:H/RL:W/RC:C/CR:X/IR:X/AR:X/MAV:L/MAC:L/MPR:L/MUI:N/MS:U/MC:N/MI:H/MA:L.

4.2. Denial of Service (DoS) Attacks

Bad actors can cause disruption to normal services, either for their own financial gain or simply to prevent legitimate transactions from occurring [25]. An example of this attack is an NFT auction in which fallback contracts can be tailored to ensure that attackers are always the highest bidder. The hypothesis is that an attacker bids with a fallback function that is designed to revert payment. If the attacker is able to become the leading bidder, then the attacker can act to make sure refunds issued to them always fail, resulting in a revert of payment. If the attacker is not able to receive the refund, no one else can call the bid function and take the spot of the highest bidder from the attacker. This denies other bidders the normal functionality of the auction and allows the attacker to always win the auction. A sample code block illustrating how such an attack can be written is shown in Figure 5. To test this condition, a smart contract instance was created in the test bed using the code shown in Figure 5. The test bed experimentally verified

that this attack is feasible during an NFT auction simulation. It is hypothesized that the attack can be mitigated by a combination of techniques, including implementing pull over push payments, marking contracts identified as untrusted, avoiding state changes after external calls, performing error handling in external calls rather than in the main contract, and not assuming contracts are created with a zero balance [25]. Each of these changes was implemented in the test bed code, and the NFT auction simulation was repeated to demonstrate that the attack was no longer effective. This process was repeated five times to validate that these recommendations worked consistently under a variety of input conditions.

```
//INSECURE
contract Auction {
        address currentLeader;
        uint highestBid;
        function bid() payable {
                require(msg.value > highestBid);
                require(currentLeader.send(highestBid)); // Refund the old
        leader, if it fails then revert
        currentLeader = msg.sender;
        highestBid = msg.value;
    }
}
```

Figure 5. Sample denial of service attack code.

Another type of DoS attack involves exploitation of gas limits. Functions inside smart contracts requre gas fees to execute computations. It is hypothesized that these fees can become excessive if a large number of small computations, or a few computationally complex actions, are taken. To test this, a new smart contract for an NFT auction simulation was deployed in the test bed, and it was shown that if the total gas fees included in a block exceed the block gas limit, the block will not be successfully written to the chain. Recommended mitigation for this type of DoS attack involves careful auditing of smart contracts for vulnerability to gas fee lmiting conditions. We verified this approach successfully mitigates attacks by implementing code which checks both transaction frequency and size, pausing execution of the contract with an error message when the established gas fees are exceeded. This process was repeated at least five times to validate that these recommendations worked consistently under a variety of input conditions.

Results of the CVSS analysis for denial-of-service attacks are shown in Figure 6, along with the corresponding vector string, following the standard approach described in [16].

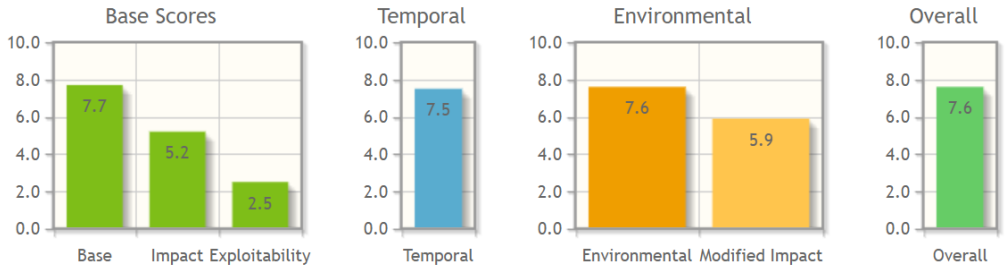

Figure 6. CVSS scores for denial of service attack; the corresponding vector string is AV:L/AC:L/PR:N/UI:N/S:U/C:N/I:H/A:H/E:H/RL:W/RC:C/CR:X/IR:H/AR:H/MAV:L/MAC:L/MPR:L/MUI:N/MS:X/MC:N/MI:H/MA:H.

4.3. Re-Entrancy Attack

When a smart contract calls another external contract, it is possible for a bad actor to make unexpected changes to the data or take over control flow (i.e., which part of the contract executes and how many times it executes) [26]. This is known as a re-entrancy attack, since it allows an attacker to re-enter the control flow in such a way that subsequent

code functions will never be executed. For example, consider a smart contract designed to make financial transactions, in which the intended victim has a contract which tracks their current balance, deposits, and withdrawals using function calls as illustrated in Figure 7. First, the attacker deposits 1 ETH (the unit of coins in Etherium) to the Victim Bank contract. This allows the attacker to pass the Require() function in the victim contract's Withdraw() function. The Attacker Contract's Attack() function then calls the victim contract's Withdraw() function. The Withdraw() function uses the built-in Ethereum Call() function to send the attack contract the value of the balance variable. Now, because Balance [Attacker] is 1, the contract sends 1 ETH to the attacker contract. However, Call() also looks to call a function in the attacker contract, and since it finds no matching signature, it automatically calls the Fallback() function. This is where the attack really begins. The Fallback() function then calls Withdraw() again, re-entering the control flow at an incorrect point. Subsequent calls to the Withdraw() function continue recursively looping until the Victim Contract is drained of all its ETH. The reason this attack was possible was because the Withdraw() function set the value of the balance [contract] back to 0 after the function call. Effectively the victim contract was tricked into believing that the attacker has a balance of 1 in its bank balance, when in reality, that value had already been sent to the attacker contract.

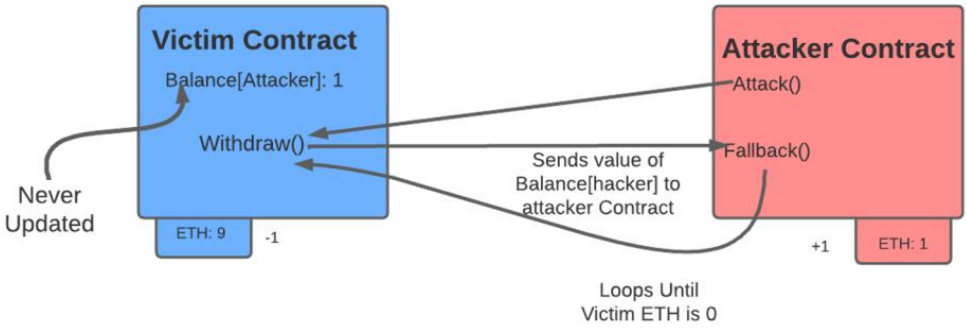

Figure 7. Control flow diagram for re-entrancy attack.

To test this attack hypothesis, we created a sample victim contract and attack script for the test bed as shown in Figure 8. Note that in order to call an external contract, the address of that contract in hex is saved in a variable and then the code will dot (.) the function inside the contract to be called. Note that the last two lines of the victim contract never execute, allowing the attacker to drain the victim's account balance. We successfully repeated this attack at least five times to demonstrate the feasibility of such an approach.

Victim Contract

```
function withdraw() public
    unit bal = balances[msg.sender];
    require(bal > 0);
    (bool sent, ) =
    msg.sender.call{value: bal}("");
    require(sent, "fail");
    balances[msg.sender] = 0;
```

Last two lines never run

Attacker Contract

```
fallback() external payable
    if(address(victim).balance >= 1 ether)
      victim.withdraw();

function attack() external payable
    require(msg.value >= 1 ether);
    victim.deposit{value: 1 ether}();
    victim.withdraw();
```

Figure 8. Sample code for re-entrancy attack.

The best practice for re-entrancy mitigation is to execute all required control flow operations inside of the function before calling an external function. We modified the smart contract code according to this recommendation, and experimentally verified that the attack was successfully mitigated. Another way to prevent re-entrancy attacks might be to use a reentrancy guard such as provided by OpenZepplin and other smart contract auditing services, although we did not test such services. Lastly, a method of preventing re-entrancy is by using a gas limit to prevent the recursive calling of a function. This can be done by using the function Transfer() which has a gas limit of 2300 specifically to prevent this type of attack. We created a new smart contract instance and tested the use of the function Transfer() to show that it successfully mitigated this attack; all mitigation tests were repeated at least five times to demonstrate effectiveness.

Results of the CVSS analysis for re-entrancy attacks are shown in Figure 9, along with the corresponding vector string, following the standard approach described in [16].

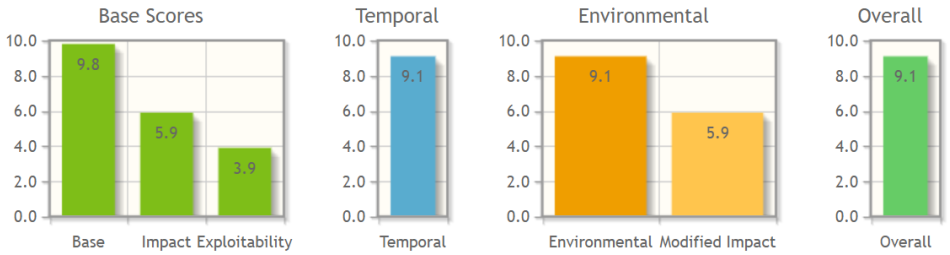

Figure 9. CVSS scores for re-entrancy attacks; the corresponding vector string is AV:N/AC:L/PR:N/UI:N/S:U/C:H/I:H/A:H/E:H/RL:T/RC:R/CR:H/IR:H/AR:H/MAV:N/MAC:L/MPR:N/MUI:N/MS:X/MC:H/MI:H/MA:H.

4.4. Cross-Chain Bridge Attack

Cross-chain bridges are used to span transactions between different blockchains. For example, an investor purchasing an NFT through layer-1 Etherium will incur relatively high gas fees. These fees are less if the purchase is made through layer-2 Arbitrum. Arbitrum must be added to the investor's Metamask wallet, then the desired amount of ETH can be converted from one chain to another. Since the bridges need to interface with multiple blockchain and smart contracts, they are a high value target and provide a unique opportunity for code vulnerabilities. For example, prior to the release of Arbitrum's Nitro code, errors were found in Solidity that prevented the Arbitrum–Etherium bridge from working correctly [27]. Due to the inbox sequencer being delayed, incoming Ethereum deposits could be diverted into an attacker's wallet without being detected. In addition, the Wormhole Bridge was attacked when the verification protocol was bypassed and a fake sysvar account was created; using the "complete-wrapped" function, a malicious message successfully minted 120,000 ETH valued at over USD 320 M [28]. Similarly, a private key leak and failure of two signature authentication allowed attackers to take control of the Harmony bridge, conscripting the multisigwallet to call the function Confirmtransaction() directly and transfer USD 100 M from the bridge [29].

As noted previously, the test bed has the capability to bridge transactions between Ethereum and Arbitrum. While the release of Arbitrum's Nitro code mitigates some forms of cross-chain bridge vulnerabilities, attacks similar to the Wormhole Bridge or Harmony Bridge are still feasible. Similar attacks can be demonstrated in the test bed. Mitigation of these attacks requires additional test and auditing of the bridge and the blockchain code on either side of the bridge, in order to identify and correct lapses in authentication or race conditions with can result in private key leakage. This can be accomplished in the test bed, although the auditing process can be time consuming and has not been automated as of this writing. Mitigation was tested at least give times to demonstrate effectiveness. Further, bridge attacks can be combined with social engineering techniques discussed previously, so mitigation must also include regular training of system administrators to resist such attacks. For example, Axie Infinity is a blockchain/NFT based online game

which uses the Ronin bridge; attackers pretending to be job recruiters targeted an employee who eventually responded to a phishing attack, giving the attackers private keys which allowed them to compromise validator nodes on the blockchain. When 5 out of 9 validators were compromised, it became possible to forge several withdrawals totaling over USD 625 M [30]. It is possible to re-create such attacks in the test bed, if it is assumed that an initial social engineering attack is successful. As before, the preferred mitigation involves regular training and education of administrators to resist these attacks.

Results of the CVSS analysis for cross-chain bridge attacks are shown in Figure 10, along with the corresponding vector string, following the standard approach described in [16].

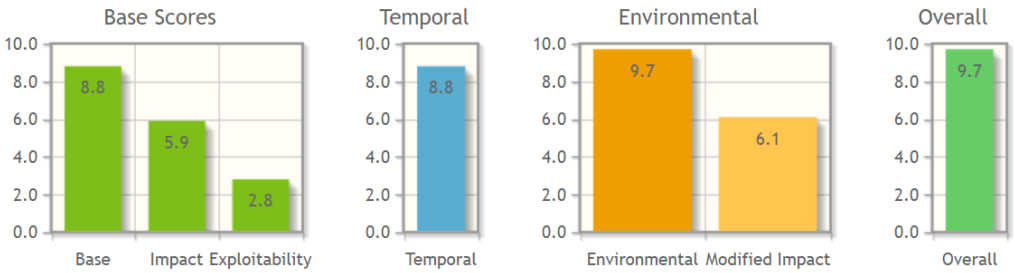

Figure 10. CVSS scores for re-entrancy attacks; the corresponding vector string is AV:N/AC:L/PR:N/UI:R/S:U/C:H/I:H/A:H/E:H/RL:X/RC:X/CR:H/IR:H/AR:H/MAV:N/MAC:X/MPR:X/MUI:X/MS:C/MC:H/MI:H/MA:H.

5. Conclusions

Blockchain, smart contracts, and related concepts have emerged in recent years as a promising technology for cryptocurrency, NFTs, and other areas. However, there are still many security issues that must be addressed as these technologies evolve. This paper reviewed some of the leading social engineering attacks on smart contracts, as well as several vulnerabilities which result from insecure code development (the code vulnerabilities can be combined with social engineering attacks in many cases). A smart contract test bed using Solidity and a Metamask wallet was used to evaluate vulnerabilities such as insecure arithmetic, denial of service, re-entrancy attacks, and cross-chain vulnerabilities resulting from layer 2 side-chain processing. Possible attack methods are hypothesized and documented and are then tested experimentally; proposed mitigations are also experimentally tested. Insecure arithmetic attacks were experimentally mitigated by changing variable data types to avoid overflow and underflow conditions, and by using the SafeMath library. Denial of service attacks were experimentally mitigated by implementing pull over push payments, marking contracts identified as untrusted, avoiding state changes after external calls, performing error handling in external calls rather than in the main contract, and not assuming contracts are created with a zero balance. Re-entrancy attacks were experimentally mitigated by executing all required control flow operations inside of the function before calling an external function, and by using the function Transfer() which has a gas limit to prevent recursive calling of a function. Cross-chain bridge attacks were the most difficult to mitigate, since they required auditing of the code on either side of the bridge to correct lapses in authentication or race conditions that can induce private key leakage.

At this time, best practices to mitigate these vulnerabilities include careful auditing of all smart contract code (with particular emphasis on order of execution exceptions), avoiding function calls untrusted contracts or insecure libraries, and avoiding transactions submitted by unvetted third parties. While the results of the testbed clearly demonstrate successful attack mitigation, the research described in this paper is limited by available data from vulnerable smart contracts. This work does not include samples of the compromised code from the breaches mentioned in the text, since this code has not yet been made publicly available. Such code samples are expected to be available in the future, and comparisons with the testbed results can be used to help ensure that the proposed approach

is correct. Future work will evaluate whether additional mitigation is required to prevent the recurrence of these attacks. In addition, the current test bed will continue to evaluate the complex and subtle code interactions which can cause security exposures for smart contracts, including a search for additional zero-day vulnerabilities not yet discovered. Additional updates or data sets are planned to be made available through the public Marist Innovation Lab Github site.

Author Contributions: Conceptualization, C.D. and B.G.; methodology, C.D. and B.G.; software, J.I., R.P., A.P. and J.V.; validation, J.I., R.P., A.P. and J.V.; writing—original draft preparation, J.I., R.P., A.P. and J.V.; writing—review and editing, C.D.; supervision, C.D. and B.G.; project administration, C.D. and B.G. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Data Availability Statement: No new data were created or analyzed in this study. Data sharing is not applicable to this article.

Acknowledgments: We gratefully acknowledge the support of Phaelan Kook and Richard Schwab during this research.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Gaur, N. "The Rising NFT Tide Lifts All Tokens", Including IBM's Definition of the Permissioned Blockchain Paradigm, April 2021. Available online: https://www.ibm.com/blogs/blockchain/2021/04/the-rising-nft-tide-lifts-all-tokens-so-what-is-an-nft/ (accessed on 8 December 2021).
2. Dixon, C. Why Web3 Matters. Available online: https://future.a16z.com/why-web3-matters/ (accessed on 8 December 2021).
3. Marlinspike, M. First Impressions of Web3. January 2022. Available online: https://moxie.org/2022/01/07/web3-first-impressions.html (accessed on 8 December 2021).
4. Jain, M.; Oliveria, M.; Shin, A.; Apostolu, D.; Wackerow, P.; Zhu, R.; Awosika, E.; Richards, S.; Zhang, L.; Cook, J.; et al. "Zero Knowledge Rollups", Etherium Documentation. Available online: https://ethereum.org/en/developers/docs/scaling/zk-rollups/ (accessed on 22 December 2022).
5. Kalodner, H.; Goldfeder, S.; Chen, X.; Weinberg, S.; Felten, E. Arbitrum: Scalable, Private Smart Contracts. In Proceedings of the 27th USENIX Security Symposium, Baltimore, MD, USA, 15–17 August 2018; pp. 1353–1370. Available online: https://www.usenix.org/conference/usenixsecurity18/presentation/kalodner (accessed on 22 December 2022).
6. Shraddha, S. Top 11 Defi Cross-Chain Bridge Attacks of 2022: Hackers Bag over $2 Billion. *BeInCrypto*. 10 October 2022. Available online: https://beincrypto.com/top-11-defi-cross-chain-bridge-attacks-of-2022-hackers-bag-over-2-billion/ (accessed on 22 December 2022).
7. Mochram, R.; Macawower, C.; Tanujaya, K.; Moniaga, J.; Jabar, B. Systematc Literature Review: Blockchain security in NFT ownership. In Proceedings of the 2022 International Conference on Electrical and Information Technology, Malang, Indonesia, 15–16 September 2022; pp. 302–306.
8. Krichin, M.; Lahami, M.; Al-Haija, Q. Formal Methods for the Verification of Smart Contracts: A Review. In Proceedings of the IEEE 15th International Conference on Security of Information Networks, Sousse, Tunesia, 11–13 November 2022; pp. 1–8.
9. Almakhour, M.; Sliman, L.; Samhat, A.; Mellouk, A. Verification of smart contracts: A survey. *J. Pervasive Mob. Comput.* **2020**, *67*, 101227–101230. [CrossRef]
10. Ante, L. Smart contracts on the blockchain: A bibliometric analysis and review. *J. Telemat. Inf.* **2021**, *57*, 101519–101525. [CrossRef]
11. Almakhour, M.; Sliman, L.; Samhat, A.; Mellouk, A. A formal verification approach for composite smart contract security using FSM. *J. King Saud Univ.-Comput. Inf. Sci.* **2023**, *53*, 70–86. [CrossRef]
12. Hu, B.; Zhang, Z.; Liu, J.; Liu, Y.; Yin, J.; Yu, R.; Lin, X. A comprehensive survey on smart contract construction and execution: Paradigms, tools, and systems. *Patterns* **2021**, *2*, 100179–100183. [CrossRef] [PubMed]
13. Hong, G.; Chang, H. A study on corporate information assets management system using NFT. In Proceedings of the IEEE 13th International Conference on Information and Communication Technology Convergence, Jeju Island, Republic of Korea, 19–21 October 2022; pp. 608–610.
14. Takahashi, H.; Lakhani, U. Sustainable NFT blockchain storage for high availability and security. In Proceedings of the IEEE 11th Global Conference on Consumer Electronics, Osaka, Japan, 18–21 October 2022; pp. 264–267.
15. Abaci, I.; Ulku, E.E. NFT based asset management system. In Proceedings of the IEEE International Symposium on Multidisciplinary Studies and Innovative Technologies, Ankara, Turkey, 20–22 October 2022; pp. 697–701.

16. Mell, P.; Spring, J.; Dugal, D.; Ananthakrishna, S.; Casotto, F.; Fridley, T.; Ganas, C.; Kundu, A.; Nordwall, P.; Pushpanathan, V.; et al. *Measuring the Common Vulnerability Scoring System Base Score Equation*; National Institute of Standards and Technology: Gaithersburg, MD, USA, 2022; NIST Internal or Interagency Report (IR) NIST IR 8409. Available online: https://nvlpubs.nist.gov/nistpubs/ir/2022/NIST.IR.8409.pdf (accessed on 13 February 2023). [CrossRef]
17. Halpern, E. What Are Zero-Knowledge Rollups, Alchemy. 2022. Available online: https://www.alchemy.com/blog/zero-knowledge-rollups (accessed on 22 December 2022).
18. Bowers, S. Langbar International, The Guardian, June 2011. Available online: https://www.theguardian.com/business/2011/jun/24/langbar-international-fraud-history (accessed on 22 December 2022).
19. Cimpanu, Law Enforcement Seizes Dark Web Market. May 2019. Available online: https://www.zdnet.com/article/law-enforcement-seizes-dark-web-market-after-moderator-leaks-backend-credentials/ (accessed on 8 December 2021).
20. Perper, R. Over $30B of NFT trading on Etherium Is Wash Trading. December 2022. Available online: https://www.coindesk.com/web3/2022/12/23/over-30b-of-nft-trading-volume-on-ethereum-is-wash-trading-research-suggests/ (accessed on 8 December 2022).
21. Martin, L. Winning the Red Queen Race. November 2022. Available online: https://www.bluetoad.com/publication/index.php?m=1336&i=659360&view=articleBrowser&article_id=3668188 (accessed on 8 December 2022).
22. Fauvel, A. The Red Queen, October 2018. Available online: https://medium.com/two-hop-ventures/the-red-queen-8d0844aa5a20 (accessed on 22 December 2022).
23. Qureshi, H. The DeFi Flash Loan Attack that Changed Everything. February 2020. Available online: https://www.coindesk.com/tech/2020/02/27/the-defi-flash-loan-attack-that-changed-everything/ (accessed on 8 December 2021).
24. OpenZeppelin Math Libraries. Available online: https://docs.openzeppelin.com/contracts/2.x/api/math (accessed on 8 December 2021).
25. Etherium Smart Contract Best Practices, "Denial of Service". Available online: https://consensys.github.io/smart-contract-best-practices/attacks/denial-of-service/ (accessed on 22 December 2022).
26. Marchenko, E. Constantinople Hard Fork Makes Us Rethink What Reentrancy Is. *Medium*, SmartDec Cybersecurity Blog. 17 January 2019. Available online: https://blog.smartdec.net/constantinople-hard-fork-makes-us-rethink-what-reentrancy-is-455716c53537 (accessed on 22 December 2022).
27. Mollen, F. Arbitrum Rewards Hacker for Detecting Critical Vulnerability. September 2022. Available online: https://bingx.com/en-us/news/20483/ (accessed on 8 December 2022).
28. Certik Whte Paper, Wormhole Bridge Exploit Incident Analysis. August 2022. Available online: https://www.certik.com/resources/blog/1kDYgyBcisoD2EqiBpHE5l-wormhole-bridge-exploit-incident-analysis (accessed on 8 December 2022).
29. Paige, C. Hacker Exploits Harmony Blockchain Bridge. June 2022. Available online: https://techcrunch.com/2022/06/24/harmony-blockchain-crypto-hack/?guccounter=1&guce_referrer=aHR0cHM6Ly93d3cuZ29vZ2xlLmNvbS8&guce_referrer_sig=AQAAAAHb0-0Gtz_bLf43LyRFT6vAgmbQ7J7BOHADplYAOAw-hqKsPz7fFW5vEVacDr3pxDDgT_xsjRuJerGXFCFQSv2IT-INVoLHlKJqv_bIU-Q3mJyaGUWr-55RDSJovfHMpexupKBoBuSZemTYg_vK3gopXpKNcpRJsGUHL7KuaVVO (accessed on 8 December 2022).
30. Colafi, A. Axie Infinity Hack. March 2022. Available online: https://www.techtarget.com/searchsecurity/news/252515336/Axie-Infinity-hack-results-in-600M-cryptocurrency-heist (accessed on 8 December 2022).