TOBB EKONOMİ VE TEKNOLOJİ ÜNİVERSİTESİ
FEN BİLİMLERİ ENSTİTÜSÜ

ETHEREUM AKILLI KONTRATLARI İÇİN GÜVENLİK DENETİM METODOLOJİSİ

YÜKSEK LİSANS TEZİ

Turgay Arda Usman

Bilgisayar Mühendisliği Anabilim Dalı
Bilgi Güvenliği

Tez Danışmanı: Prof. Dr. Ali Aydın Selçuk

OCAK 2022

ÖZET

Yüksek Lisans/Doktora Tezi

ETHEREUM AKILLI KONTRATLARI İÇİN GÜVENLİK DENETİM METODOLOJİSİ

Turgay Arda Usman

TOBB Ekonomi ve Teknoloji Üniversitesi
Fen Bilimleri Enstitüsü
Bilgisayar Mühendisliği Anabilim Dalı
Bilgi Güvenliği

Danışman: Prof. Dr. Ali Aydın Selçuk

Tarih: Ocak 2022

Akıllı kontratlar blokzincir teknolojisi altında çalışan, belli amaçları gerçekleştirmek için yazılmış küçük programlardır. Blokzincir teknolojisinin en temel özelliklerinden olan değişmezlik ilkesi akıllı kontratlar için de geçerlidir. Bu da zincire yüklenmiş olan bir kontratın değiştirme ve manipülasyonlara karşı dirençli olduğu anlamına gelir. Bu kontratlarda bulunan mantık hatalarının ve zafiyetli kısımların zincire yüklendikten sonra değiştirilemeyeceği anlamına da gelmektedir. Saldırganlar için bu durum büyük bir kolaylıktır. Bunu önlemek için akıllı kontrat projeleri bloğa yüklenmeden önce üstlerinde güvenlik denetimleri uygulanmaktadır. Ancak kimi zaman bu denetimler de yetersiz kalır ve projeler saldırganların hedefi olur.

Bu çalışmada Ethereum ekosisteminde meydana gelmiş olaylardan önemli etkiye sahip olanları incelenerek projelerin denetimlerinde bir eksik varsa bunu tespit etmek ve buna ek olarak ekosisteminde sıkça rastlanan zafiyetleri inceleyerek kapsamlı bir denetim metodolojisi önermek ve bu sayede bahsi geçen olayların bu metodoloji ile denetlenen kontratların başına gelmesinin önlenmesi amaçlanmıştır.

Ethereum ekosistemindeki daha önceden denetlenmiş ve önemli saldırıların hedefi olan projelerin denetleme raporları incelendiğinde, büyük çoğunluğunun önüne geçebilmenin mümkün olduğu görülmüştür. Bunun nedeninin ise söz konusu projelerin işletme mantıkları incelenirken gözden kaçan noktalar ve sadece otomatik araçlara güvenen süreçler olduğu gözlemlenmiştir. Buna ek olarak çalışmada

incelenen zafiyetlerde etkin zafiyetlerin %60'ının kullanıcı ihmallerinden kaynaklandığı görülmektedir. Bu tarz zafiyetlerin tespiti mantıksal eksikliklere göre daha kolaydır, hem araçlar hem insanlar için.

Tüm anlatılanlar göz önüne alındığında bu çalışmada önerilen metodolojinin kabaca manuel analiz ile otomatik araçları bir arada kullanmayı hedeflediğini söylemek makuldür. Manuel analiz yöntemlerini esas olarak kabul eden metodoloji hem statik hem dinamik manuel analiz sırasında dikkat edilmesi gereken noktaları ve bu noktaların nasıl incelenmesi gerektiğini anlatır. Otomatik araçları ise manuel analiz yöntemlerinin eksik kaldığı, uç nokta testleri, gözden kaçan noktalar, hatalı pozitif veya negatiflerin doğrulanması gibi noktalarda kullanır. Bu sayede iki yaklaşımın birbirlerini eksiklerini kapatacak şekilde kullanılması sağlanmış olur ve kapsamlı bir denetim sağlanır.

Anahtar Kelimeler: Blokzincir, Ethereum, Akıllı kontrat, Denetim, Zafiyet.

ABSTRACT

Master of Science/Doctor of Philosophy

THESIS TITLE

Turgay Arda Usman

TOBB University of Economics and Technology
Institute of Natural and Applied Sciences
Computer Science Programme
Information Security

Supervisor: Prof. Dr. Ali Aydın Selçuk

Date: January 2022

Smart contracts are small programs that work under blockchain technology and are written to achieve certain purposes. The principle of immutability, which is one of the most fundamanetal features of blockchain technology, also applies to smart contracts. This means that a contract deployed to the chain is resistant to modification and manipulation. However, this means that the logic errors and vulnerabilities in the contracts cannot be fixed after they are uploaded to the chain. For attackers, this is a great convenience. To prevent this, security checks are applied to smart contract projects before they are uploaded to the block. However, sometimes these controls are insufficient and projects become the target of attackers.

In this study, it is aimed to examine the events that have a significant impact on the Ethereum ecosystem, to detect if there is a deficiency in the audits of the projects, and to suggest a comprehensive audit methodology by examining the frequently encountered vulnerabilities in the ecosystem, and thus to prevent the mentioned events from happening to the contracts audited with this methodology.

When the audit reports of the projects in the Ethereum ecosystem that were previously audited and the target of important attacks are examined, it has been seen that it is possible to prevent the vast majority of them. It was observed that the reason for this was the points that were overlooked while examining the business logic in question and the processes that only rely on automated tools. In addition, it is seen that 60% of the active vulnerabilities in the vulnerabilities examined in the study are

caused by user negligence. Such vulnerabilities are easier to detect than logical ones, both for vehicles and humans.

To sum up, it is reasonable to say that the methodology proposed in this study aims to combine manual analysis with automated tools. The methodology, which accepts manual analysis methods as its basis, explains the points to be considered during both static and dynamic manual analysis and how these points should be examined. It uses automated tools where manual analysis methods are lacking, such as endpoint testing, overlooked points, and verification of false positives or negatives. In this way, it is ensured that the two approaches are used in a way that makes up for each other's deficiencies and a comprehensive audit is provided.

Keywords: Blockchain, Ethereum, Smart contract, Audit, Vulnerability.

TEŞEKKÜR

Çalışmalarım boyunca değerli yardım ve katkılarıyla beni yönlendiren hocam Prof. Dr. Ali Aydın Selçuk ve Dr. Süleyman Özarslan'a, kıymetli tecrübelerinden faydalandığım TOBB Ekonomi ve Teknoloji Üniversitesi Bilgisayar Mühendisliği Bölümü öğretim üyelerine, destekleri ve bilgi birikimleri ile her zaman yanımda olan kıymetli akıl hocam Aytek Yüksel ve Onur Alanbel'e, aile ve arkadaşlarıma çok teşekkür ederim.

İÇİNDEKİLER

Sayfa

ŞEKİL LİSTESİ

Sayfa

ÇİZELGE LİSTESİ

Sayfa

KISALTMALAR

ETH **:** Ether
EVM **:** Ethereum sanal makine
API **:** Uygulama programlama ara yüzü
ABI **:** İkili Uygulama ara yüzü
PRNG **:** Sözde rastgele sayı üreteci
DeFi **:** Merkezi olmayan finans

1. GİRİŞ

Akıllı kontratlar günümüz blokzincir teknolojisinin en çok tercih edilen yapı taşlarındandır. Bunun altında yatan sebeplerden en önemlileri sağladıkları otonomi ve ek güvenlik olarak gösterilebilir. Akıllı kontratları belirli amaçlar için yazılmış, blokzincirdeki küçük programlar olarak düşünmek mantıklı olacaktır. Blokzincir teknolojisi içerisinde kullandıkları için bu teknolojinin en temel özelliklerinden biri olan değişmezlik ilkesini de sağlamaktadırlar. Bu da demek oluyor ki zincirde bulunan bir kontratta değiştirme veya manipülasyon yapmak mümkün değildir. Bu durum her ne kadar kontrat kodunun dışarıdan müdahalelere karşı dirençli olmasını sağlasa da kontrat yaratılırken yapılmış olan mantık hatalarının da düzeltilmesinin imkansız olduğu anlamına gelmektedir.

Ethereum günümüzde Bitcoin'den sonra en çok bilinen blokzincir teknolojisidir. Ancak Bitcoin'den farklı olarak sadece kripto para birimi olarak işlev göstermek yerine, çok daha fazlasını kullanıcılarına sunmayı amaçlar. Desteklediği akıllı kontrat yapısı ve durum bazlı makine anlayışı (Ethereum Sanal Makinesi gibi) sayesinde merkezi olmayan uygulamaların da geliştirilmesine ve çalıştırılmasına olanak tanır. Bu sayede blokzincirin günümüz teknolojik anlayışında daha fazla yer bulmasına olanak sağlamıştır.

Bu tezin akışı şu şekildedir: takip eden bölümde blokzincir konsepti ve kullandığı teknolojilerden bahsedilmiştir. Daha sonra akıllı kontratların yapısı ve Ethereum hakkında bilgi verilmiştir. Devamında ise daha önce de bahsi geçmiş olan önemli hack olayları incelenmiştir. Üçüncü bölümde ise Ethereum platformunda bulunan çeşitli güvenlik zafiyetleri detaylı bir biçimde incelenmiştir. Dördüncü bölümde akıllı kontratların denetimi için kapsamlı bir denetim (audit) metodolojisi önerilmiştir. Son olarak dördüncü bölümde çalışmanın sonuçları ve ilerleyen süreçte çalışmanın evirilebileceği potansiyel yönelimler bulunmaktadır.

1.1 Tezin Amacı

Her teknolojide olduğu gibi blokzincir teknolojisinde de güvenlik zafiyetleri bulunmaktadır. Bu zafiyetlere her geçen gün yenileri eklenmektedir. Blokzincirin yapısı gereği değiştirilemez olması da saldırganların işini kolaylaştırmaktadır. Gözden kaçan herhangi bir mantık hatası, zafiyetli kod kullanımı zincirdeki kontratı direkt bir hedef haline çevirir. Bu tarz durumların yaşanmasını önlemek için denetim (audit) adı verilen denetimler geliştirilmiştir. Bir denetimi bir uygulamaya yapılan bir sızma testine benzetmek yanlış olmayacaktır. Denetim servisi sağlayan pek çok firma bulmak mümkündür. Bu firmaların hepsinin kendine ait bir metodolojisi bulunmaktadır. Ancak her ne kadar denetim servisleri büyük bir titizlikle uygulansa da yine de denetim edilmiş bir kontrat veya merkezi olmayan uygulamanın saldırganların kurbanı olması mümkündür. Bu tez çalışmasının ana amacı yukarıda sözü edilen olaylardan ses getirmiş olanları inceleyerek eğer bu projelerin denetimlerinde bir eksik varsa bunu tespit etmek, buna Ethereum ekosisteminde sıkça rastlanan zafiyetleri incelemesini de ekleyerek daha kapsamlı bir denetim metodolojisi önererek, ilerleyen zamanda bu tip olayların sayısını azaltmaya çalışmaktır. Aslına bir nevi güvenlik denetimleri için dikkat edilmesi gereken noktaları içeren standart bir metodoloji ortaya koymaktır. Bu çalışmanın sonucunda ortaya çıkan metodolojiden bahsetmek gerekirse; metodoloji, manuel analiz yöntemlerini esas olarak kabul etmektedir. Manuel analizin, yan insanın eksik kaldığı noktaları otomatik araçlar yardımı ile ortadan kaldırmayı amaçlar. Bu eksik noktalara hatalı pozitif ve negatif değerleri, tam kapsanamayan uç nokta testleri örnek verilebilir. Bu amaçla önerilen metodoloji, manuel analiz yöntemlerini esas olarak kabul etmektedir. Otomatik araçları (statik analiz ve gri kutu fuzzlama araçlarının kullanılması yapılan çalışmalar sonucu önerilmektedir.) ise manuel analiz yöntemlerinin eksik kaldığı uç nokta testleri, hatalı negatif ve pozitif değerlerin tespiti, matematiksel formül doğrulamaları gibi noktalarda kullanmaktadır. Bu sayede kapsamlı bir denetim süreci oluşturmak mümkün hale gelmiş olur.

1.2 Literatür Araştırması

Literatürde akıllı kontralarda bulunan zafiyetler üzerine yapılmış pek çok çalışma bulunmaktadır. Ancak akıllı kontratlarda güvenlik denetimlerinin nasıl yapılması

gerektiği ile ilgili çalışmalar yok denecek kadar azdır. Bu tez çalışması bu alandaki eksikliğin giderilmesine yardımcı olmayı amaçlamaktadır.

Atzei, Bartoletti ve Cimoli [1], Ethereum'un güvenlik açıklarını analiz etti ve bu açıklardan yararlanan güvenlik açıkları ve saldırılar için bir sınıflandırma sağladı.

Pariteeshan ve ark. [2]., Ethereum akıllı sözleşmelerindeki bazı ana güvenlik açıklarını belirledi. 16 güvenlik açığını ayrıntılı olarak araştırdılar ve bunları 19 yazılım güvenlik sorunuyla ilişkilendirdiler. Aynı zamanda pek çok statik analiz, dinamik analiz ve resmi doğrulama aracını da incelediler. Bu sayede güvenlik açıklarının yanı sıra analiz araçlarını ve onların kısıtlarını da ortaya koydular.

Samareen ve Alalfi [3], sekiz güvenlik açığına odaklandı ve bunların istismar senaryoları ve farklı araçlardaki tespit oranları da dahil olmak üzere ayrıntılı bir analiz sağladı.

Sayeed, Marco-Gisbert, ve Caira [4], blokzincirdeki istismar tekniklerini ve en yaygın yedi saldırıtekniğini inceledikleri çalışmalarında, bu teknikleri aynı zamanda sınıflandırmışlardır. Yapılan çalışma aynı zamanda piyasada yaygın olarak kullanılan 10 güvenlik aracının bu teknikler üzerindeki sonuçlarını incelemiş ve her ne kadar bu araçlar çok çeşitli çözümler sunsa da bu atakların hala etkili olduğunu görmüşlerdir.

Perez ve Livshits [5], altı akademik çalışmayı baz alarak yaptıkları çalışmalarında 23,327 zafiyetli kontratı incelemişlerdir. Bu çalışmada bu kontratların sadece %1,98'inin gerçekten zafiyetli olduğunu tespit etmişlerdir.

Cao, Zou, ve Cheng [6], *Flashot* adında bir prototip önerdiler. Bu prototip akıllı kredi işlemi sırasında varlık akışlarını şeffaf bir şekilde gösterebilmektedir. *Flashot* özellikle Arbitraj ve Pump vakalarını incelemektedir. Bu incelemelerde saldırganın davranışlarına yönelik derinlemesine ekonomik açıklamalar sunulmuş ve saldırılarının gelişim eğilimlerini çıkarmak amaçlanmıştır.

Kaihua Qin, Liyi Zhou, Benjamin Livshits, ve Arthur Gervais [7], ROI'leri %500 binin üzerinde olan iki mevcut saldırıyı analiz ettikleri çalışmalarında, saldırı parametrelerini bulmayı, temeldeki Ethereum blokzincirini durumu ve DeFi ekosisteminin durumu üzerinde bir optimizasyon problemi olarak formüle etmeyi amaçlamışlardır.

Bhardwaj ve ark. [8], akıllı kontratlar ve merkezi olmayan uygulamalar için sızma testi yapısı önerdikleri çalışmalarında, önerdikleri yaklaşımı otomatik araçlar ile karşılaştırmışlarıdır. Elde ettikleri sonuçlarda bu yeni yaklaşımın ilgili alan için alışılagelmiş sızma testi yaklaşımlarına göre daha etkili olduğunu göstermişlerdir.

2. ARKA PLAN

2.1 Blokzincir

Blokzincir veya diğer adıyla dağıtık kayıt defteri yapısı, gerçekleşen işlem kayıtlarının şifrelenmiş bir şekilde saklandığı, gelişmekte olan dağıtık bir sistem olarak tanımlanabilir [9]. Blokzincir, üç temel bileşenin bir araya gelmesi ile oluşur, blok, zincir ve ağ. Blokzincirde kayıtlar eş (*peer)* adı verilen bağlantı noktalarında saklanır. Eşlerin bir araya gelmesi ile blok adı verilen yapılar oluşur. Bir blokta maksimum işlem sınırına ulaşıldığında o blok, kendinden önce gelen blokların bulunduğu zincir adı verilen yapıya dahil edilir. Ağ ise, dağıtık işlem kayıtlarından oluşan bağlantı noktalarının bir araya gelmesi ile oluşur.

Blokzincir yapısı gereği, dışarıdan yapılan müdahalelere karşı dirençlidir. Bunun sebebi blokzinciri teknolojisini benimsemiş yapıların büyük çoğunluğunun blok oluşumu sırasında, işlem kayıtlarını saklarken genellikle Merkle-Patricia ağacı yapısı gibi yapıları tercih etmesidir. Merkle-Patricia ağacında yeni işlem kayıtları en aşağıdaki dalları oluşturur. Bloklar da bu işlem kayıtlarının bir araya gelmesi ile oluşur ve her ağacın sadece kök bileşenini barındırır. Zincirler de blokların bir araya gelmesi ile oluştuğu için, blokzincir üzerinde yapılmak istenilen herhangi bir değişiklik zincirde kırılmaya yol açar.

Blokzincir, dağıtık yapısına uygun bir şekilde Bizans (*Byzantine)* tehdit modelini benimser. Normalde merkezi yapıya sahip sistemlerde, veriye erişmek için izin alınması gerektiğinde, bu yetkilendirmeyi yapacak olan ve bu sebeple güvenilmesi gereken yapıların bulunması gerekmektedir. Bu da güvenilen kaynağın compromise olması gibi çeşitli güvenlik açıklarına yol açabilir. Ancak dağıtık yapısı sayesinde blokzincirde veriye erişmek için herhangi bir güvenilen kaynağa ihtiyaç yoktur, geliştiriciden kullanıcıya herkes güvenilmez olarak kabul edilebilir. İşte bu yaklaşımda Bizans Tehdit Modeli olarak adlandırılır.

2.1.1. Uzlaşma algoritmaları

Bir önceki bölümde anlatılanlar baz alınarak blokzincir merkezi olmayan bir veri tabanı olarak düşünülebilir. Buradan yola çıkarak, merkezi olsun veya olmasın, tüm veri tabanlarında çakışmalar ve tutarsızlıklar ile karşılaşmak mümkündür. Bu çakışmalar ve tutarsızlıklar olarak anlatılan sorunlara Bizans Arızaları (Byzantine Fault) adı verilmektedir. Bizans Arızaları genelde dağıtık sistemlerde görülür, sistemde meydana gelen bir değişikliğin sistemdeki tüm bileşenlerde doğru şekilde algılanamaması durumlarında meydana gelirler. Blokzincir için Bizans Arızaları yapılan bir işlemin sistemdeki dağıtık kayıt defterleri tarafından doğru bir biçimde kayıt altına alınamaması olarak anlatılabilir.

Blokzincir, Bizans Arızaları gibi problemlerin üstesinden gelebilmek için uzlaşma algoritmalarını kullanır. Uzlaşma kelimesi bir konu üzerinde tarafların hepsinin aynı görüşte birleşerek uzlaşma sağlaması anlamına gelmektedir. Buradan yola çıkarak bir uzlaşma algoritmasının işlevi sistemde gerçekleşen bir işlemin tüm sistem bileşenleri tarafından aynı şekilde kayıt altına alınmasının sağlanması olarak düşünülebilir. Bu blokzinciri dışarıdan müdahalelere karşı dirençli yapan yapılardan bir tanesidir. Bu başlığın kalanında popüler uzlaşma algoritmalarının birkaçından detaylı olarak bahsedilecektir.

Emek Kanıtı (Proof of Work)

Bu yaklaşım belki de en popüler uzlaşma algoritmalarının başında gelmektedir. Bitcoin ve Ethereum gibi büyük blokzincirler bu yaklaşıma güvenmektedirler. Ancak her ne kadar bu yaklaşım yaygın bir biçimde kullanılıyor olsa da enerji tüketimi ve hesaplama bakımından oldukça pahalıdır. Bu pahalılığın sebebi emek kanıtı konseptinin çalışma prensibinden kaynaklanmaktadır. Bu konseptin çalışma mantığı şu şekildedir: Her bloğun bir zorluk derecesi vardır, bu dereceler belli sayıda sıfıra ile başlayan değerlerdir. Emek Kanıtı yaklaşımı, bir madenciden (miner) blok içeriği ile birleştirilip özet fonksiyonu alındığında bu zorluk derecesine eşit olacak bir değer bulmasını ister. Bu hesaplama işlemine madencilik (mining), bu işlemi gerçekleştiren bileşenlere de madenci adı verilir. Bu hesaplama zincir üzerinde yapılacak tüm değişikliklerden önce hesaplanmak zorundadır. Bu sayede sistem dışarıdan müdahalelere karşı bir nebze olsun korunmuş olur.

Ancak her yöntemde olduğu gibi bu yöntemde de çeşitli zayıflıklar mevcuttur. Bunlardan en önemlisi %51 saldırısı olarak da bilinen bir saldırıya yol açan bir mantık zafiyetidir. %51 atakları saldırganın blokzincirde yukarıda bahsedilen özet fonksiyon alma işleminde kullanılan gücün %50'den fazlasına, genellikle %51, tekabül eden bileşenleri veya bileşen gruplarını ele geçirmesi ile blokzincirde gerçekleşen işlemleri kontrol etmeye başlamasıdır.

Hisse Kanıtı (Proof of Stake)

Bir diğer bilinen yaklaşım da Hisse kanıtı konseptidir. Bu konsept her ne kadar çalışma prensipleri açısından tartışmalara yol açsa da Ethereum yeni versiyonu olan Ethereum 2.0'da bu konsepti benimseyeceğini açıklamıştır. Çalışma mantığını özetlemek gerekirse; sistemde belli bir değerin üzerinde Ether'i bulunan doğrulayıcılar arasından rastgele bir seçim yapılır. Seçilen bu doğrulayıcı yeni blokları üretir ve bu sayede blok ödüllerini kazanır.

Yukarıda bahsedilen çalışma prensibinden de anlaşılabileceği gibi bu yaklaşımda hesaplama gücü ön planda değildir. Ethereum özelinde konuşmak gerekirse 32 Ether'i hisse (stake) olarak sunabilecek kişiler doğrulayıcı olmaya hak kazanır. Doğrulayıcıların görevi yeni blokların oluşmasını sağlamak veya kendi yaratmadıkları blokları kontrol etmektir. Doğrulayıcıların sundukları 32 Ether'lik hisse değeri yaptıkları işlemlere göre azalıp artabilir. Bu duruma örnek olarak bir doğrulayıcının erişilemez hale gelmesi durumunda sunduğu 32 Ether'lik değerin bir kısmını kaybetmesi verilebilir. Bu mantıkta rastgele seçilen bir doğrulayıcının ürettiği bir bloğu geri kalan doğrulayıcıların doğrulayacağını söylemek mümkündür. Bu yapı doğrulayıcıların birbirleri ile yarışmak yerine bir arada çalışmasına olanak sağlar.

Bu yaklaşımda dikkat edilmesi gereken bir diğer kavram da işaret zinciri (beacon chain) denilen bir yapıdır. Ethereum, Hisse kanıtı yaklaşımını benimsemesi ile sistem üzerinde parçalama (sharding) yapacağını duyurdu. Böylece Ethereum 64 parçalık blokzincirlerden oluşacak şekilde çalışacak. Bu zincirlerin hepsinde yukarıda anlatılan şekilde Hisse kanıtı algoritması çalışacak. Bu düzeni kontrol ve koordine etmek için ise işaret zinciri denilen yapı kullanılacak. İşaret zinciri her bir parçadan durum bilgisini alarak bunu tüm sistem için erişilebilir hale getirecek. Aynı zamanda doğrulayıcıların kaydolması, ödül ve cezaların tahsili gibi işlemler ile de ilgilenecek.

Bu zincir başta Ethereum Mainnet'ten ayrı bir yapı olacaksa da zamanla ikisi birbirine entegre olacak [10]. Aynı zamanda bu sayede Ethereum daha iyi bir ölçeklendirme anlayışı benimseyebilecek.

Hisse kanıtı yöntemi emek kanıtına göre her ne kadar daha enerji ve hesaplama gücü açısından daha efektif ve daha iyi ölçeklendirme olanağı sunuyor olsa da. Yine de %51 atakları gibi saldırıları yapmak mümkündür. Burada dikkat edilmesi gereken nokta %51 atağını gerçekleştirmenin büyük ölçüde zorlaşmış olmasıdır. Bunun nedeni, artık bir saldırganın %51 atağını gerçekleştirebilmek için sistemdeki işletim gücünün %51'ini değil de doğrulayıcıların yatırmış olduğu giriş meblağlarının %51'ini ele geçirmiş olması gerektiğidir. Bu meblağ hem çok yüksek bir değere tekabül ettiğinden bu atağın meydana gelme olasılığı hayli düşmektedir. Aynı zamanda bu saldırının en fazla etkileyeceği kişi bu miktarı elinde tutan kişi olacaktır, çünkü böyle bir saldırı dolaşımda olan Ether değerinde bir düşüş, bir devalüasyon meydana gelmesine yol açacaktır. Bu da Ether sahiplerinin para kaybetmesi anlamına gelir ki zaten saldırıyı gerçekleştirebilmek için yüklü miktarda Ether'e sahip olmak gerekiyor. Bu sebepler göz önünde bulundurularak, böyle bir saldırının meydana gelme olasılığının çok düşük olduğunu belirtmek yanlış olmayacaktır.

2.1.2. Blokzincir türleri

Blokzincir teknolojisinin dört ana türü bulunmaktadır.

Herkese açık blokzincir

Halka açık blokzincir teknolojisi, bu teknolojinin bugün en yaygın olan ve en çok kullanılan türüdür. Bitcoin, Ethereum, Avax, Solana bunlara örnek olarak verilebilir. Bu tür blokzincirlere, isminden de tahmin edilebileceği, gibi herkes katılıp işlem gerçekleştirebilir veya madencilik (miner) yapabilir.

Özel blokzincir

Adından da anlaşılabileceği gibi herkesin katılamayacağı, dışarıya kapalı bir blokzincir yapısıdır. Genellikle üyeleri davet veya benzeri protokoller ile kabul edilir. Bilinen blokzincir yapısının aksine genellikle merkezi denilebilecek bir yapıya daha yakındır. Ancak içeride akan veri şifrelenmiştir. Bu türe örnek olarak *Multichain* verilebilir.

Konsorsiyum blokzincir

Bu tür blokzincirler, önceden belirlenmiş bir grup eleman (node) tarafından idare edilir. Zincir üzerinde işlem yapabilme hakkı blokzincirin dizaynına göre herkese açık veya sınırlandırılmış olabilir. Bu tarz mimariler genellikle bankalar veya benzeri kurumlarda kullanılır. Corda ve Quorum bu tarz zincirlere örneklerdir.

Melez blokzincir

Bu yaklaşım özel ve herkese açık blokzincir yaklaşımlarının birleşiminden oluşur. Bununla anlatılmak istenilen şey bu yaklaşımda zincirin üyeleri zincire kimin katılacağından hangi işlemlerin herkese açık olacağına, kimin hangi işlemleri yapma yetkisine sahip olduğuna kadar her şeyi özelleştirme hakkına sahiptirler. Kısacası bir özel blokzincir yapısında herkese açık blokzincir özelliklerinin sergilenmesine olanak tanımaktadır.

2.2 Akıllı Kontrat

Blokzincirdeki yapı taşları olan işlemlerin otomatik bir şekilde gerçekleştirilmesine olanak sağlayan yapılara akıllı kontrat denilmektedir. Blokzincir 2.0 ile ortaya çıkan bu yapılar, günümüzde yaygın ve aktif bir biçimde kullanılmaktadır. Akıllı kontratlar bahsi geçen otomasyonu sağladıkları programlı yaklaşım ile sağlarlar. Akıllı kontrat geliştirirken kullanılan en yaygın programlama dilleri Solidity ve Vyper'dır. Akıllı kontratların en çok bilinen kullanım yeri Ethereum alt yapısıdır.

Akıllı kontratlar Ethereum Sanal Makine (EVM) gibi altyapılarda çalışan Turing complete yapılardır. Bu yapılar blokzincir üzerinde rahatça çalışabildikleri için merkezi bir otoriteye ihtiyaç duymazlar. Aynı zamanda Turing complete yapıları sayesinde merkezi olmayan uygulamaların (DAPP) da ortaya çıkmasına olanak sağlarlar. Akıllı kontratlar hakkında unutulmaması gereken en önemli şey, bu yapıların bir kere yayınlandıktan sonra üzerlerinde oynama veya değiştirme yapmanın mümkün olmadığıdır. Bu sayede blokzincirin değiştirilemez yapısına uyum sağlarlar.

Akıllı kontratlar bugün başta finans sektörü olmak üzere, enerji ve nesnelerin interneti (IoT), gibi pek çok alanda kullanılmaktadır. Finans sektöründe akıllı kontratların kullanılması ile sektördeki şeffaflık artmış, altyapısal maliyetler azalmıştır [2].

2.3 Ethereum

Ethereum blokzincir teknolojisini sistemdeki değişiklikleri saklamak ve sistem içindeki yapı taşlarını senkronize etmek için kullanmaktadır. Bunu yaparken de işlem maliyetlerini takip ve manipüle etmek için Ether adı verilen kripto para biriminden faydalanır. Proje dokümanında (whitepaper) Ethereum'un amacı şu şekilde tanımlanmıştır: Ethereum'un amacı, merkezi olmayan uygulamalar oluşturmak için alternatif bir protokol oluşturmak ve hızlı geliştirme süresi, küçük ve nadiren kullanılan uygulamalar için güvenlik sağlanması, farklı uygulamaların birbirleriyle efektif iletişim kurması gibi yönlere ağırlık veren ödünleşimler sağlamaktır [11]. Ethereum bu tanımlanmış amacına akıllı kontratları kullanarak ulaşmayı amaçlamaktadır. Bu sayede daha önce de bahsedilen merkezi olmayan uygulamaları destekleyebilir hale gelmiştir, hatta bu uygulamalara yerleşik ekonomik özellikler eklenmesine de olanak sağlamıştır. Buraya kadar yapılan açıklamalardan da anlaşılabileceği gibi Ethereum'un amacı bir kripto para birimi olmak değil, kripto para birimini bir araç olarak kullanan çok amaçlı dağıtık bir sistem olabilmektir.

2.3.1. Ethereum Sanal Makinesi (EVM)

Ethereum'un akıllı kontrat gibi kompleks bir yapıyı destekleyebilmesi için akıllı kontratların çalışabileceği bir altyapı sunması gerekmektedir. Ethereum bu altyapıyı Ethereum Sanal Makinesi (EVM) adı verilen yapı ile sağlar. EVM makina kodu ve programlama dili arasında bir soyutlama katmanı olarak çalışır. Bu sayede akıllı kontrat var olan programlama dili seçenekleri arasından herhangi biri ile yazılabilir. Yazılan kontratların hepsi aynı makine koduna dönüştürüleceği için bir sorun çıkmayacaktır.

EVM yığın tabanlı bir sanal makine olarak çalışır. Bu yığın 1024 eleman tutabilme kapasitesine sahiptir. Bu yığın içerisinde durum (state) bilgisi saklanır. Bu saklama Merkle-Patricia ağacı yapısı kullanılarak gerçekleştirilir. Bu yapı şu şekilde fonksiyon göstermektedir; her eleman (node) kendisinin iki çocuğunun özet fonksiyon bilgisinden oluşacak şekilde yaratılır, en alttaki elemanlar ise çocukları henüz olmadığından direkt olarak veri saklar. Bu yapıya sahip EVM'de 140 tane opcode bulunmaktadır [12]. Bu opcode'lar EVM'in Turing complete bir durum makinesi (state machine) gibi çalışmasına olanak tanır.

2.3.2. İkili Uygulama Ara yüzü (ABI)

Ethereum ekosisteminde gerçekleşen bir işlemde, işlemin kaynağı, istikameti, hedefte hangi kontrat fonksiyonları ile etkileşeceği ve gerekli argümanlar bulunmaktadır. Bu bilgiler ikili uygulama ara yüzü (ABI) denilen bir yapı şeklinde taşınır. Bu sayede Ethereum ekosistemi içerisindeki kontratlarla hem ekosistem içinden hem de ekosistemin dışından iletişim kurulabilir.

2.3.3. Gaz

Bu çalışmada daha önce de bahsedildiği gibi Ethereum gerçekleşen işlemlerin maliyetlerini takip ve manipüle etmek için Ether gibi yapıları kullanmaktadır. Bu cümleyi daha da açmak gerekirse, Ethereum ekosisteminde bir işlemin gerçekleşmesi için önden gerekli maliyetlerin ödenmesi gerekmektedir. Bu amaçla ödenen miktarlara *Gaz* adı verilir. Ancak burada unutulmaması gereken nokta Gaz yapısının Ether'den farklı olduğudur. Ether'in değeri bulunduğu pazara bağlı iken Gaz değerleri marketten bağımsızdır. Gaz değeri tüm işlemler için şu şekilde hesaplanır [13];

- Gas_maliyeti x Gas_ücreti
    - Gas_maliyeti: *Gaz Limiti* olarak da bilinir. Maksimum Gaz maliyetini temsil eder.
    - Gas_ücreti: işlemi başlatan parti tarafından belirlenir. İşlemi başlatanların ödemeyi kabul ettikleri maksimum Gaz değerini temsil eder.

Bu formül ile yapılan hesaplama sonucunda iki senaryo ortaya çıkar.

1. **Gaz Tükenmesi İstisnası (Out-of-Gas Exception):** Gaz ücretinin, Gaz maliyetini geçtiği durumlarda, gerçekleşmekte olan işlem derhal sonlandırılır ve o zamana kadar yapılan ilgili tüm işlemler geri çevrilir. Ancak harcanmış olan Gaz değeri harcandığı hali ile kalır.
2. **Gaz Fiyatının çok Yüksek veya Düşük Olması:** Gaz ücretinin çok düşük olması hiçbir madenci (miner) tarafından işlemin değerlendirilmemesine, çok yüksek olması ise işlemi başlatan taraf için gereksiz derecede bir maliyetin ortaya çıkmasına yol açar.

2.3.4. Anlık Kredi (Flash Loan)

Anlık kredi yapısı Ethereum'da son zamanların en çok kullanılan özelliklerindendir. Geleneksel finans alanında tam olarak bir karşılığı bulunmasa da bu yapıyı, geleneksel terimleri kullanarak açıklamak gerekirse kısa süreli teminatsız kredi olarak tanımlamak mantıklı olacaktır. Anlık kredi yapısında belirtilen süre içerisinde (genellikle o işlem sonunda) aldıkları miktarı geri ödeyebildikleri sürece kullanıcılara büyük miktarlarda borçlanma olanağı tanınır. Şayet borç verilen miktar belirtilen sürede ödenmezse yapılan işlem geri çevrilir.

Anlık Kredilerin kullanım alanlarından bazıları aşağıdaki gibidir [14].

- **Arbitraj**: Bir varlık için farklı fiyat sağlayan platformlar arasında ticaret yaparak fayda elde edinilmesi olarak tanımlanabilir. Anlık krediler sayesinde alıcılar ellerinde hazır varlık olmadan alım satım yapabilirler. Bu sayede eğer alıcı Gaz ücretini karşılayabilirse masrafsız bir biçimde varlık sahibi olabilir.
- **Kendi Kendine İşlem (Wash Trading):** Bir varlık veya platform için sahte işlem hacmi yaratmaktır. Bu anlık krediler ile çok yüksek sermayelere sahip olmadan da yapılabilir hale gelmiştir.
- **Teminatlı Swap (Collateral Swap)**: Kullanıcının kredisini destekleyen teminatın başka bir teminat türüyle hızla değiştirilmesi.

2.4 Vaka İncelemesi

Çalışmanın bu bölümünde, Ethereum ekosistemi üzerinde son zamanlarda gerçekleşmiş büyük etki yaratan saldırıların raporları ve bu protokollerin saldırılardan önce yapılmış olan denetim raporları incelenecektir. Ayrıca yapılmış denetimlerde izlenen metodoloji ve bu metodolojinin gözden kaçırdıkları tartışılacaktır.

- StableMagnet en çok bilinen *rugpull* [72] olaylarından biridir. Totalde 27 milyon dolarlık bir kayba neden olan bu olay, pek çok açıdan ilginçtir. Buna Etherscan ve BSBScan gibi platformların kontratın kaynak kodunu doğrulamaması örnek olarak gösterilebilir. Böyle bir doğrulamanın yapılmamış olması, kodun iddia edilen şeyleri yapıp yapmadığından emin olunamayacağı anlamına gelir. 27 milyon dolarlık kayıp da tam olarak bunun

yüzünden olmuştur. Projede bulunan *SwapUtil* isimli bir kütüphane aslında zararlı kod içermekteydi, bu kütüphane bir nevi arka kapı görevi görmektedir. Aynı zamanda bu kütüphanede bulunan kod tüm sermayeyi çekmeye olanak tanımaktaydı [15]. Bu protokol Techrate [16] adı verilen bir firma tarafından denetlemiştir. Firmanın ilgili protokole yapmış olduğu denetim raporu dipnotta bulunabilir [73]. Bu rapor incelendiğinde raporun otomatik statik analiz araçları ile hazırlandığı görülebilir. Bu sebeple, bu tarz zararlı bir kodun gözden kaçırılması çok doğaldır.

- Harvest [17], 25 milyon dolarlık bir kaybın yaşandığı bir olaydır. Bu vakanın en önemli özelliklerinden biri, akıllı kredinin arbitraj özelliği kötüye kullanılarak hedefin sömürülmüş olması. Bu olayın gerçekleşmesine yol açan iki önemli faktör var; Bunlardan ilki projenin gerçek zamanlı değerleri temel hesaplamalarda kullanması, diğeri ise bazı kasaların içindeki varlıklar, temel *DeFi* protokollerinin ortak havuzlarına yatırılıyordu. Bu tür havuzlardaki varlıklar, kalıcı kayıp, arbitraj ve kayma gibi piyasa etkilerine tabidir. Böylece, büyük hacimli piyasa işlemleri yoluyla bu değerleri manipüle etmek mümkündür [18]. Bu proje hacklenmeden önce iki farklı şirket tarafından denetlenmiştir, bu şirketlerin ilgili raporlarına dipnotta ulaşılabilir [74]. Bu raporlara bakıldığında iki şirketin de gayet detaylı raporlar çıkardığını söylemek mümkün. Bu denetimlerin ikisinde de arbitraj kontrolü yapan *depositArbCheck* isimli fonksiyonun mantıksal hataya sahip olduğu bu sebeple doğru çalışmadığı dile getirilmiştir. Bu fonksiyonun düzeltilmiş olması atağı önlemeye yardımcı olabilirdi.
- Grimm Finance, bir yield farming (verim çiftliği) projesidir [19]. Bu proje Aralık 2021 tarihinde bir saldırı kurbanı olmuştur. Bu saldırıda projedeki *depositFor*() isimli fonksiyonda bulunan reentrancy zafiyeti sömürülmüştür. Bu sayede saldırgan zararlı transferler yaparak kasadaki hissesini yükseltmeyi başarmıştır. Bu projenin denetimi Solidity Finance tarafından yapılmıştır ve detaylı rapora dipnotta ulaşmak mümkündür [75]. Firma denetim işlemi sırasında bu zafiyeti keşfedememiştir. Bunun sebebi olarak da CTO değişimi ve hızlı büyüme dolayısıyla gözden kaçırılma olarak gösterilmiştir. Sunulan rapor incelendiğinde bu raporun hazır bir analiz aracı çıktısı olduğunu söylemek çok da yanlış olmayacaktır.

- Furucumbo, kullanıcılarına kar ettirmeyi amaçlayan bir kripto portal olarak çalışmaktadır [20]. Bu projeyi kullanan kişilerin, kullanım sırasında belirli seviyede dışarıya erişim izni vermesi gerekmektedir. Aslında bu projenin kurban olmasının sebebi de bu erişim iznidir. Saldırganlar projenin vekil kontratında bulunan argümanları kullanıcı tarafından belirlenen iç içe *delegatecall* çağrılarını kötüye kullandılar [21]. Bu projenin denetim raporları incelendiğinde aslında *delegatecall* kullanımıyla ilgili pek çok sorunun bulunduğu görülmektedir, bunların büyük çoğunluğu işletme mantığının tam olarak kodda uygulanamamasından kaynaklanmaktadır. Spesifik bir alanda bu kadar zafiyetin ortaya çıkması, belli zafiyetlerin gözden kaçmalara yol açmış olabilir.

Sonuç olarak bakıldığında, bu çalışmada önerilen metodoloji ile burada incelenen olayların büyük bir kısmının önüne geçilebilir. Tabii, Harvest protokolünde olduğu gibi dile getirilen sorunun düzeltilmemesine bir şey yapmak mümkün değil. Yine de sadece araçlara bağlı kalan veya işletme mantığı ile kaynak kodun bağdaşmadığı durumlar önlenebilir.

3. ETHEREUM GÜVENLİK ZAFİYETLERİ

Çalışmanın bu bölümünde Ethereum ekosisteminde yaygın olan zafiyetler incelenecektir. Böyle bir inceleme ile amaçlanan şey denetim işlemi sırasında bir denetimcinin (auditor) karşılaşabileceği zafiyetler hakkında genel olarak bir bilgi edinmesini sağlamaktır. Bu zafiyetlerin güncelliğini sağlamak için bilinen belirli zafiyetlere (reentrancy, blok değişkenlerinin kullanımı gibi) ek olarak 2020 yılında en çok karşılaşan zafiyetler için de bir çalışma yapılmıştır. Ortak sonuçları bu başlığın ilerleyen bölümlerinde incelemek mümkündür.

2020 yılında en çok karşılaşılan zafiyetleri tespit etmek için Evengy Medvedev'in [22] hazırlamış olduğu ve sürekli olarak güncel tuttuğu açık veri seti kullanılmıştır. Veri setinde 2020 yılı için filtreleme yapıldığında 15 milyon civarında kontrat verisi elde edilmiştir. Elde edilen kontrat verisinin analiz edilebilmesi için kaynak kodları temin etmek istendiğinde Etherscan API sınırlamaları ile karşılaşılmıştır. Bu sebeple veri seti üzerinde dağıtım uygulanmıştır. Bu sayede her ayda 90,000 özgün kontrat olacak şekilde 1 milyon 80 bin kontrat elde edilmiştir. Bu kontratların analizi için statik analiz yöntemi seçilmiştir. Bunu iki nedeni vardır. Bunlardan ilki statik analizin muadillerine göre daha hızlı sonuç verebilmesi diğeri ise çalışmanın bu kısmı yeni zafiyet keşfetmektense bilinen zafiyetleri tespit etmeye odaklı olması ve statik analiz yöntemlerinin bu konudaki olgunluğunun muadillerine göre daha fazla oluşudur. Slither [23] statik analiz aracını kullanarak hazırlanmış bir kod Amazon Web Servisleri üzerine yüklenmiştir. Burada altı c5a.4xlarge makine eş zamanlı olarak veri setinin belirli kısımları için yazılan kodu çalıştırmıştır. Slither aracının seçilmesindeki en önemli neden, aracın sağlamış olduğu geniş filtre ve test koşullarının çalışmanın yapıldığı zamanda güncel ve yeterince kapsamlı olmasıdır. Kısacası araç incelenecek zafiyetleri tespit edebilecek filtre ve koşulları ve daha nicelerini hali hazırda barındırmaktadır. Aynı zamanda bu test ve filtrelerin çeşitli varyasyonları tespit edebilecek kadar olgunlaşmış olduğunu da belirtmek gerekir.

3.1 Zafiyetler

Analiz sonucu çıkan zafiyetler bu bölümde incelenmiştir. Yapılan incelemeler, zafiyet hakkında detaylı bilgi, zafiyetin gerçek hayatta kullanıldığı önemli senaryolar ve bu zafiyetin önlenmesi için önerileri içermektedir.

3.1.1. Reentrancy

Reentrancy, bir kontrattaki fonksiyon veya fonksiyonların, kontrat durumunun güncellenmesine izin vermeden sürekli olarak çağrılması olarak tanımlanabilir. Burada kontrat durumunun güncellenmesinden kastedilen, herhangi bir fonksiyon çalıştıktan sonra yapılan değişikliklerin kontrata yansıması olarak düşünülebilir. Bu güvenlik açığını kötüye kullanan saldırı teknikleri, bir kontratı tamamen yok edebilir veya tüketebilir. Bunlara ek olarak, reentrancy'den yararlanan bir saldırgan, kurbanın akıllı kontratının kontrolünü ele geçirebilir [4] [3]. Bu güvenlik açığından yararlanmak için bir saldırganın harici bir akıllı kontrattaki bir veya daha fazla fonksiyona birden çok kez erişmesi gerekir. Dolayısıyla bu güvenlik açığı adından da anlaşılacağı gibi özyinelemeli bir saldırı ile istismar edilebilir. Reentrancy zafiyeti iki genel kategoride incelenebilir: tek bir fonksiyonda reentrancy ve fonksiyonlar arası reentrancy.

Tek fonksiyonda reentrancy

Tek bir fonksiyon üzerinde reentrancy yapmanın mantığı, dışarıdan kontrata erişilirken harici olarak işaretlenen bir fonksiyonu veya *fallback()* fonksiyonunu ilk çağrı tamamlanmadan önce sürekli olarak çağırmaktır [24]. Bu, başlangıçta çağrılan fonksiyonun yürütülmesini bozacak ve saldırganın kontratı istediği gibi manipüle etmesine izin verecektir. Sömürü sırasında en çok *create()* ve *fallback()* fonksiyonları tercih edilir. Bunun nedeni, bu fonksiyonların etkisidir.

***fallback()* Fonksiyonu Bazlı Reentrancy**: *fallback()* fonksiyonu, herhangi bir argümanı olmayan ve hiçbir şey döndürmeyen harici bir ödenebilir fonksiyondur. Kontrat, herhangi bir veri dahil olmadan Ether aldığında veya kontratta mevcut olmayan bir fonksiyonu çağırmaya ihtiyaç duyulduğunda yürütülür [4]. Solidity, başka bir kontratı çağırmak için SEND ve CALL olmak üzere iki yerleşik yapıya sahiptir. Her ikisi de CALL komutlarını kullanır ve varsayılan bir Gaz değeri olmadan çalışır. Bir kontrat, Ether'i başka bir kontrata aktarmak için SEND

kullandığında, alıcı sözleşmede *fallback*() fonksiyonunu başlatır. Saldırganın *fallback*() fonksiyonunun içeriğini istediği gibi değiştirebileceği ve gaz limitinin olmadığı göz önüne alındığında, reentrancy zafiyeti olan bir sözleşmeyi boşaltmak saldırgan için artık kolaydır.

Create() Fonksiyonu Bazlı Reentrancy: CREATE, her biri bir STOP komutu tarafından takip edilen fonksiyonlarının başlatma kodlarıyla yeni bir kontrat oluşturan bir komuttur [1]. Bu yeni oluşturulan kontratlar güvenilirdir ve anında yürütülür. Bu nedenle, bu kontratların yaratıcılarının (constructor) kötü niyetli kontratlara çağrı yapmasını engelleyecek hiçbir mekanizma yoktur. Çağrılan kötü niyetli kontrat, hedef kontrata istekte bulunmaya başlayabilir ve ondan fon çekilebilir.

Fonksiyonlar arası reentrancy

Şimdiye kadar tek bir fonksiyon kullanan reentrancy saldırılarından bahsedilmiştir, ancak aynı tutarsız durumda iki fonksiyonu kullanarak reentrancy güvenlik açığından yararlanmak mümkündür. Daha doğrusu saldırgan, her seferinde aynı dahili değişkeni kullanan farklı fonksiyonlara erişmeye çalışarak o değişkeni manipüle etmeyi amaçlar. Fonksiyonlar arası reentrancy kötüye kullanan saldırılar, genellikle kendilerini gizlemek için üçüncü taraf kitaplıkları kullanır.

3.1.1.1. Bilinen ataklar

Bu güvenlik açığından yararlanan bilinen iki büyük saldırı vardır; DAO saldırısı ve Spankchain saldırısı. DAO, bir kitle fonlaması platformu sözleşmesiydi. Saldırgan, işlemler bir mecburi çatallanma (hard fork) [25] tarafından bloke edilene kadar yaklaşık 60 milyon dolar ele geçirdi. Güvenlik açığı, DAO kontrol listesindeki *SplitDAO* fonksiyonundan kaynaklanıyordu. Bu fonksiyon, yukarıda açıklanan *fallback()* fonksiyonu mantığına karşı savunmasızdır. Hedef fonksiyon akışının sonunda dengesini ve durumunu günceller; bu nedenle, başka bir sözleşme bu fonksiyonu güncellemeden önce çağırırsa, saldırganın istediği kadar parayı yinelemeli olarak çekebilir.

SpankChain bir yetişkin eğlence platformudur. 2018'de bir reentrancy saldırısının kurbanı oldu ve 65 Ether kaybetti. Maliyetleri en aza indirmek ve verimliliği artırmak için sözleşmelerinde ERC20 desteğini kullanıyorlardı [26]. Ancak saldırıya kadar kullandıkları bu kütüphanenin gerçekliğini kontrol etmediler. Kullandıkları

fonksiyonlardan biri, kötü amaçlı kod içeren sahte bir fonksiyondu. Bu yüzden fonksiyonlar arası bir reentrancy saldırısının kurbanı oldular.

3.1.1.2. Öneriler

Şu ana kadar yapılan analizlere göre reentrancy'e karşı korunmanın iki yolu vardır; üçüncü taraf kütüphanelerin kimliğini doğrulamak ve harici fonksiyonları çağırmadan önce dahili durum değişiklikleri gerçekleştirmektir (Checks-Effects-Interactions deseni [27] olarak da bilinir). Bunlara ek olarak mutex lock (reentrancy lock) kullanımının etkili bir yöntem olduğu gösterilmiştir. Openzeppelin'in ReentrancyGuard'ı bu tür yeniden giriş kilitlerine bir örnektir [28].

3.1.2. Block.timestamp değişkeni

Zaman damgaları, diğer herhangi bir teknolojide olduğu gibi, Ethereum blokzincirde zamana bağlı çeşitli işlemler için kullanılmaktadır. Rastgele sayı üretimi ve belirli bir süre için fon kilitleme bu işlemlere örnektir. Ethereum'da zaman damgası işlemleri *block.timestamp* değişkeni aracılığıyla gerçekleştirilir. Ancak *block.timestamp'te* blok altyapısından kaynaklanan bir mantık hatası var.

Her blok, Ethereum blokzincirde bir zaman damgası içerir. Bu zaman damgaları genellikle madencinin (miner) sistem zamanına ayarlanır. Bir blok kazıldığında (mine), bir madenci Emek kanıtı bulmacasının tamamlanma süresini gösteren bir zaman damgası üretmelidir. Ancak madenciler bir öncekinden daha küçük bir zaman damgası ayarlayamazlar ve alınan bloğun zaman damgasından 900 saniyeden daha büyük olmayan bir zaman damgası ayarlamaları gerekir [2][29].

Bahsedilen mantık, saldırganların zaman damgalarını istedikleri gibi manipüle etmelerini sağlar. Bu, bir saldırganın hedeflenen sözleşmeyi manipüle etmek için izin verilen zaman aralığı içinde bir zaman damgasını yeniden düzenleyebileceği veya farklı bir zaman damgası seçebileceği anlamına gelir. Bunu yaparak bir saldırgan, kontratın davranışını kendi çıkarları doğrultusunda değiştirebilir. Bu, bir saldırganın belirli bir süre boyunca rastgele sayı oluşturma ve para birimi kilitleme gibi işlemleri manipüle edebileceği anlamına gelir.

3.1.2.1. Bilinen ataklar

GovernMental [30] kontratına yapılan saldırı, *block.timestamp* güvenlik açığını kötüye kullanan en iyi bilinen saldırıdır. Kontratın doğrulanmış kaynak kodu

0xF45717552f12Ef7cb65e95476F217Ea008167Ae3 [15] adresinde bulunabilir. Bu kontrat esasen bir piramit şemasıdır, ancak tur tabanlı bir piyango oyununa benziyor. Bu kontratı sömürmenin başka birçok yolu olmasına rağmen, bu madde yalnızca zaman damgasına odaklanır. GovernMental, geçerli saati belirlemek için *block.timestamp* kullanır. Bir saldırgan, 12 saatin sonuna yakın bir andaki zaman damgasını bir sonraki zaman damgasıyla değiştirebilirse, turu erken bitirebilir ve tüm Ether'i alabilir. Benzer şekilde, daha düşük bir zaman damgası ayarlayarak, bir saldırgan turun daha uzun sürmesini sağlayarak daha fazla para birikmesine neden olabilir.

3.1.2.2. Öneriler

En etkili düzeltme önerisi, *block.timestamp* ve eşdeğerlerini (*now, StartTime, EndTime,* vb.) kullanmamaktır. Ayrıca blok zaman damgası yerine kahin (*oracle)* gibi yapıların kullanılması daha uygun olacaktır [29]. Ancak *block.timestamp* kullanılacaksa, rastgele sayı üretiminde veya sözleşmenin Ether kilitleme gibi karar verme mekanizmalarında kesinlikle *block.timestamp* kullanılmamalıdır.

3.1.3. Katı eşitliklerin*/assert* kullanımı

Solidity akıllı kontratlarında katı eşitliklerin veya *assert* kullanılması, hizmet reddi (Denial of Service) saldırıları dahil çok sayıda saldırıya yol açabilir. Sözdizimi bakımından ve anlamsal olarak hiçbir sorun yoktur, ancak güvenli kodlama açısından katı eşitliklerin veya *assert* kullanımı geniş bir saldırı yüzeyi oluşturur.

Bu güvenlik açığından yararlanmak için kullanılan mantık şu şekilde çalışır; Bir kontratta Ether göndermek veya durum hakkında karar vermek gibi kritik durum değişikliklerini kontrol etmek için katı bir denklem veya *assert* kullanılırsa, bu koşulun gerçekleşmesini engelleyerek kontratın davranışı manipüle edilmeye çalışılır. Burada, kontratın davranışını manipüle etmekle kastedilen, kontratı boşaltmak, *fallback*() fonksiyonlarını değiştirmek, kendi kendini yok etmeyi (*selfdestruct*) veya eşitlik Ether gönderirken kullanılıyorsa hizmet reddi saldırısı tetiklemektir.

3.1.3.1. Bilinen ataklar

Gridlock hatası kötüye kullanan saldırı, katı eşitliklerin/*assert* kullanılmasının nasıl sorunlara yol açabileceğinin en bilinen örneğidir. Bu hata, Edgeware blokzinciri

platformunun Lockdrop kontratlarında meydana geldi. Aslında, katı eşitlikler güvenlik açığı ile zincirlenmiş bir hizmet reddi güvenlik açığıdır. Bu hata, saldırganların 900 milyon dolar değerinde Ether'i [31] hedeflemesine neden oldu. Savunmasız sözleşme, 0x1b75B90e60070d37CfA9d87AFfD124bB345bf70a [32] adresinden gerçekleştirilebilir.

Edgeware sözleşmesinin kilit işlevinde onaylamanın kullanıldığı bir satır var;

assert(address(lockAddr).balance == msg.value);

Buradaki *assert* programsal olarak doğru görünebilir, ancak bir saldırgan, kontrat henüz oluşturulmamış olsa bile, bir sonraki satırdaki *lock* adresine (lockAddr) Ether göndermek için bunu kötüye kullanabilir. Böylece, bir sonraki *lock* adresini önceden hesaplayarak ve bir başlangıç tutarı göndererek, bir saldırgan kontratı gelecekteki tüm *assert*'lerde başarısız olmaya zorlayabilir [31]. Bu saldırıda aynen böyle oldu.

3.1.3.2. Öneriler

En etkili çözüm katı denklemlerden ve assert kullanımından kaçınmaktır [33].

3.1.4. Rastgele hedeflere Ether gönderme

Ether'i rastgele hedeflere göndermek aslında kullanıcı tarafından oluşturulan bir güvenlik açığıdır. Genellikle kullanıcının kontratı hazırlarken gerekli kontrol mekanizmalarını eklememesi veya kullanmaması sonucu görülür. Kullanıcıların kontratlardan para çekmelerini sağlayan fonksiyonlar, akıllı kontratlarda oldukça yaygındır. Bu tür fonksiyonların kötüye kullanılmasını önlemek için yukarıda belirtilen kontrol mekanizmaları gereklidir. Şekil 3.1'deki kod parçacığı, kavramı daha iyi anlamaya yardımcı olacaktır.

```
contract ArbitrarySend{
    address destination;
    function setDestination(){
        destination = msg.sender;
    }
    function withdraw(){
        destination.transfer(this.balance);
    }
}
```

Şekil 3.1: Rastgele Gönderme Kod Parçacığı [34]

Yukarıda görülebileceği gibi, *withdraw()* fonksiyonunun herhangi bir miktarda mevcut Ether'i istenen hedefe göndermesini engelleyecek hiçbir kontrol mekanizması yoktur. Bunu kötüye kullanmak için bir saldırganın tek yapması gereken, bir alıcı belirlemek için önce *setDestination()* fonksiyonunu çağırmaktır. Ardından saldırgan, istenen miktarda Ether elde etmek için *withdraw()* fonksiyonunu çağırabilir.

3.1.4.1. Öneriler

Daha önce de belirtildiği gibi, bu güvenlik açığı kullanıcı ihmalinden kaynaklanmaktadır. Bu tür durumları önlemek için Solidity'de belirli yerleşik kontrol mekanizmaları vardır. Solidity'nin bu tür kontrol mekanizmalarına örnek olarak *required* ifadesi ve *modifier* yapıları verilebilir.

3.1.5. Kilitlenmiş Ether

Kilitli Ether güvenlik açığından mustarip olaylar, çoğunlukla düşmanca davranıştan ziyade kullanıcı hatalarından kaynaklanır. Basitçe söylemek gerekirse, kilitli Ether'de fonlar, Ethereum blokzincirdeki bir geçersiz adrese gider ve kullanılamaz hale gelir. Bu güvenlik açığının sayısız nedeni olmasına rağmen, bu güvenlik açığının saldırgan değeri çok düşüktür. Bunun nedeni, bir saldırganın yalnızca bu güvenlik açığını kötüye kullanarak fonları dondurabilmesidir. Bahsi geçen nedenlerden birkaçı aşağıda belirtilmiştir.

Kontratta bir ödeme yöntemi tanımlanmış olsa da (*fallback()* fonksiyonu gibi), CALL, DELEGATECALL, SELFDESTRUCT gibi ödeme işlemleri yapabilen sistem çağrıları yoktur [35]. Bir kullanıcı, Solidity kaynak kodunda veya Web3 test senaryolarında hedef kontratın adresini bir şekilde kaçırarak, 0x0 adresine fon aktaracaktır. Tahmin edilebileceği gibi, 0x0'daki fonlar kilitlenir ve kullanılamaz hale gelir [36]. Dış kütüphanelerin ve kod parçalarının kullanımı Solidity geliştirme topluluğunda çok yaygındır. Böyle bir durumda, herhangi bir nedenle, örneğin sözleşmenin kendi kendini imha etmesi (selfdestruct) durumunda, harici sözleşme kullanılamaz ve bu da fonların dondurulmasına neden olur [5]. Bunlara ek olarak önceden sonlandırılmış (selfdestruct) bir kontrat adresine veya geçersiz bir adrese fon yollamak da Ether kilitlenmesine yol açar.

3.1.5.1. Bilinen ataklar

Bu güvenlik açığı, ihmalin bir sonucu olduğundan, bilinen saldırılardan bahsetmek yerine, bu bölümde Kilitli Ether kurbanlarının bir listesi sunulacaktır [36]:

- 1.040.773 $ değerindeki EOS token'ları, EOS Token Akıllı Sözleşmesinde kilitlendi.
- 1.255.279$ değerindeki Enigma token'ları ENG akıllı kontratında kilitlendi.
- 1.167.192 $ değerindeki ZeroX token'ları, ZRX akıllı kontratında kilitlendi.

3.1.5.2. Öneriler

Birincil öneri, CALL, DELEGATECALL, SELFDESTRUCT sistem çağrılarından yararlanan ve Ether aktarma kabiliyetine sahip yöntemlere adres kontrollerinin dahil edilmesi olacaktır.

3.1.6. Eksik sıfır adres (0x0) doğrulaması

Sıfır adres doğrulaması eksikliğinin ardındaki mantık ve etki, kilitli Ether'e benzer. 0x0'a gönderilen Ether'ler dondurulduğu gibi, *modifier* ve fonksiyonlar gibi kontratın mülkiyetinin değiştiği durumlarda, bir adres belirtilmemişse, kontratın mülkiyeti de 0x0 olarak değişir. Bu durumun nedenini daha iyi anlamak için sıfır adresinin nasıl çalıştığını açıklamak mantıklı olacaktır. Sıfır adresi aslında bir bloğa ilk kez bir kontrat yüklenirken kullanılan 20 baytlık özel bir adrestir. Bu şekilde EVM, bu işlemin yeni bir kontratı devreye almak olduğunu anlar ve buna göre hareket eder. Aslında sıfır adresi, EVM'in çalışmasına yardımcı olmak için kullanılan bir tür sahte adrestir [37].

Bilinen ataklar

Bilinen bir saldırı yerine burada örnek bir saldırı senaryosu açıklanacaktır: Çizlege 3.2'deki kod parçası incelendiğinde *updateowner* fonksiyonunun aldığı argümanda kontratın sahipliğini *address* değişkenine verdiği görülmektedir. Ancak burada önemli olan nokta, bu fonksiyon argüman olmadan çağrılırsa sözleşmenin sahipliğinin 0x0'a geçeceğidir.

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

Şekil 3.2: Eksik Sıfır Adres (0x0) Doğrulaması Kod Parçacığı [34]

3.1.6.1. Öneriler

Kilitli Ether güvenlik açığı gibi, birincil öneri, geliştiricileri sıfır adres kontrolleri uygulamaya veya varsayılan bağımsız değişken değerlerini kullanmaya teşvik etmek olacaktır.

3.1.7. Yerel değişken gölgeleme

Gölgeleme, özellikle ayırma gibi gaz tüketimini azalttığı işlemlerde, Solidity'de yasal ve yararlı bir tekniktir. Bununla birlikte, gölgeleme sorunlara da neden olabilir. Gölgeleme sorunlarının çoğu durum değişkenlerinin gölgelenmesinden kaynaklansa da yerel değişkenlerin gölgelenmesi de sorunlara neden olabilir. Örnek olarak çizelge 3.3'deki kod parçası verilebilir:

```
pragma solidity ^0.4.24;
contract Bug{
  uint owner;
    function sensitive_function(address owner) public{
        //..;
        require (owner == msg.sender);
    }
    function alternate_sensitive_function() public{
        address owner = msg.sender;
        //..;
        require (owner == msg.sender);
    }
}
```

Şekil 3.3: Yerel Değişken Gölgeleme Kod Parçacığı [34]

Bu kod parçacığında, *owner* değişkeninin, fonksiyonlardaki *owner* değişkenleri tarafından gölgelenmesi mümkündür. İlk bakışta, bunun bu makaledeki diğer güvenlik açıkları gibi büyük bir etkisi yok gibi görünüyor. Ancak fonksiyonlarda

owner değişkenlerinden biri kullanıldığında yanlış *owner* değişkeni tespit edilecek ve istenmeyen davranışların ortaya çıkması kaçınılmaz olacaktır.

3.1.7.1. Öneriler

Gölgeleme tamamen adlandırma ile ilgili olduğundan, aynı ada sahip değişkenlerin değiştirilmesi, bunu önlemek için bir geliştiricinin her değişken için farklı adlar kullanması gerekir.

3.1.8. Kullanılmayan CALL çıktıları

Bu güvenlik açığı, Solidity'nin Ether gönderme yöntemindeki anlayıştan kaynaklanmaktadır. Ether işleme yeteneğine sahip olan *call, send, delegecall* gibi yöntemler bulunmaktadır. Hatalı bir durum veya başarısızlık durumunda, bir istisna atmak yerine, bu işlevler bir boolean değeri olan *FALSE* değerini döndürür [37]. Bu durum problem yaratır. Normalde işlemde bir hata olduğunda Solidity işlemi geri alır fakat burada Solidity bir hata veya istisna ile karşılaşmadığından işlem bozulmaz veya geri alınmaz. Böylece beklenmedik sonuçlar ortaya çıkabilir. Saldırgan, bu beklenmedik davranışlardan yararlanmak için yukarıda belirtilen fonksiyonları kullanan kontratları hedefleyebilir.

3.1.8.1. Bilinen ataklar

Bu güvenlik açığını kötüye kullanan bilinen iki büyük saldırı vardır. Birincisi, aslında oyuncuların kazanmak için belirli bir ücret karşılığında işlem yapmasını gerektiren basit bir oyun olan King of the Ether Throne olarak bilinen saldırıdır. Sözleşme yeni bir krala para gönderdiğinde, gaz ücreti olarak istemeden 2300 gaz alır. Bu miktardaki gaz ücreti bir Ethereum Mist cüzdanı için yeterli değildir ve bu nedenle işlem başarısız olur. Ancak King of Ether Throne bu işlemi *send* fonksiyonu ile gerçekleştirmektedir. Yukarıda belirtildiği gibi, işlemde bir sorun varsa, *send* fonksiyonu bir istisna atmaz, bu nedenle işlem devam eder. Bu davranış saldırganlar tarafından suiistimal edildi ve önceki kralın kazancını kontratta kilitli bıraktı, böylece sadece kontrat sahibi ona erişebildi [38].

İkincisi EtherPot olarak bilinir, esasen bir piyango sözleşmesidir. Buradaki ana güvenlik açığı *block.hash* değişkenlerinin yanlış kullanımı olmasına rağmen, kullanılmayan CALL dönüş değerleri güvenlik açığı da etkin rol oynamaktadır. Kontratın savunmasız kısmı şekil 3.4'de görülebilir.

```
function cash(uint roundIndex, uint subpotIndex){

    var subpotsCount = getSubpotsCount(roundIndex);

    if(subpotIndex>=subpotsCount)
        return;

    var decisionBlockNumber =
getDecisionBlockNumber(roundIndex,subpotIndex);

    if(decisionBlockNumber>block.number)
        return;

    if(rounds[roundIndex].isCashed[subpotIndex])
        return;
    //Subpots can only be cashed once. This is to prevent double payouts

    var winner = calculateWinner(roundIndex,subpotIndex);
    var subpot = getSubpot(roundIndex);

    winner.send(subpot);

    rounds[roundIndex].isCashed[subpotIndex] = true;
    //Mark the round as cashed
}
```

Şekil 3.4: *EtherPot* Zafiyetli Kod Parçacığı [39]

99. satırda *send* fonksiyonu kullanılıyor. Daha önce belirtildiği gibi, bu fonksiyon herhangi bir istisna oluşturmaz, bunun yerine *boolean* değerleri döndürür. Yukarıda görüldüğü gibi *send* fonksiyonu için herhangi bir kontrol bulunmamaktadır. Böylece, yine, bu fonksiyon, daha önce belirtildiği gibi saldırgan tarafından kötüye kullanılabilir.

3.1.8.2. Öneriler

Her şeyden önce geliştiriciler işlem yapmak istediklerinde *transfer* fonksiyonunu tercih etmelidirler. Ancak, *send* veya *call* gibi fonksiyonların kullanılması gereken durumlar varsa, olası bir hata durumunda dönüş değerini kontrol etmek ve işlemi durdurmak için kontroller eklenmelidir [40]. Başka ve daha güvenilir bir çözüm, bir para çekme modeli (withdrawal pattern) benimsemektir. Para çekme modelinde, her kullanıcı sözleşmeden Ether gönderilmesini yöneten ve başarısız gönderme işlemlerinin sonuçlarıyla ilgilenen izole bir geri çekme (withdraw) fonksiyonu çağırmalıdır [37].

3.1.9. Zayıf rastgele sayı üretimi (PRNG)

Bir kavram olarak rastgelelik, Ethereum için bir sorundur. Bunun nedeni, Ethereum'un deterministik bir yapıya sahip olmasıdır. Yani, Ethereum ortamındaki

her işlem ağın geri kalanı tarafından doğrulanmalıdır ve bloğun durumu buna göre güncellenir. Bu yapı rastgelelik sorunlarına yol açar, Ethereum ekosisteminde çok sayıda piyango ve kumarla ilgili sözleşmeler mevcut olduğundan bu ironik bir durumdur. Tahmin edilebileceği gibi, bu sözleşmeler çoğunlukla tamamen rastgelelik üzerine kuruludur.

Ethereum'daki rastgele veri oluşturma yöntemlerinin büyük çoğunluğu, saldırganlar için saldırı yüzeyleri oluşturur. Rastgele sayı/veri oluşturma yöntemlerinin en yaygın ve savunmasız yöntemlerinden bazıları aşağıda listelenmiştir:

- **Blok değişkenlerine dayalı PRNG'ler**: *block.timestamp, block.difficulty, block.coinbase, block.number* ve daha birçok blok değişkeni, rastgele sayı üretimi sırasında kaynak olarak kullanılır. *block.timestamp'in* ayrıntılı bir incelemesi bu bölümün başlarında bulunabilir. Kalan değişkenler genellikle benzer şekilde kötüye kullanılır. Kısacası, tüm blok değişkenleri madenciler tarafından bir şekilde manipüle edilebilir. Bu, saldırganların uygun koşulları sağlaması durumunda hedefin kullanacağı aynı rastgele değeri kullanabilecekleri anlamına gelir.
- **Blok özet fonksiyonlarına dayalı PRNG'ler:** Her blok için doğrulama amacıyla blok özet fonksiyonu mevcuttur ve EVM bunları *blok.blockhash* fonksiyonu aracılığıyla alır. Bu fonksiyon bir blok numarasını girdi olarak kabul eder ve böylece aynı rastgele değeri tahmin etmek veya elde etmek mümkündür. Bu manipülasyon üç şekilde yapılabilir: İlk olarak, mevcut blok numarası girdi olarak iletilebilir. Bloğun mevcut blok özet fonksiyonu, yürütme zamanında katılımcıları engellediği bilinmemektedir, bu nedenle sıfır değeri döndürür. İkinci olarak, başka bir yaklaşım, bloğun son özet fonksiyonunu girdi olarak geçirmektir. Kontratı kullanarak bir PRNG'den yararlanmak için dahili bir kontrat kullanılıyorsa, bu yaklaşım nedeniyle aynı rastgele sayıya sahip olacaklardır. Son olarak, gelecekteki bir bloğun blok özet fonksiyonu kullanılabilir. Ancak, EVM yalnızca en son 256 bloğa erişime izin verdiği için. Bir sonraki çağrı, saldırganların önceden sözde rasgele numara almasına izin verecektir [41].

3.1.9.1. Bilinen ataklar

block.timestamp bölümünde bahsedildiği gibi GovernMental olayı da blok değişkenlerinin nasıl kötüye kullanılabileceğinin bir örneği olabilir. Blok özet fonksiyonlarına gelince, SmartBillions piyango olayı buna iyi bir örnektir. SmartBillions kontratı, rastgelelik oluşturmak için son özet fonksiyonunu kullanır. Yukarıda belirtildiği gibi, bu yaklaşım savunmasız olabilir. Saldırganlar bunu fark etti ve hackathon başlamadan önce 400 Ether ele geçirdi [42].

3.1.9.2. Öneriler

block.hash değişkenlerinin kullanımı, blok değişkenlerin kullanımından nispeten daha güvenlidir, ancak daha önce belirtildiği gibi, hepsi saldırılara karşı savunmasızdır. Ethereum'da rastgele veri üretmenin daha güvenli bir yolu, *kahin* kullanmaktır. *Kahin* yapıları, Ethereum ortamı ile dış dünya arasında bir köprü görevi gören yapılardır. Böylece Ethereum ortamı zincir dışı bilgileri kullanabilir.

3.1.10. Tanımlanmamış yerel değişkenler

Solidity, değişkenleri depolama (storage) ve bellek (memory) olmak üzere iki farklı biçimde saklar. Depolama değişkenleri, fonksiyon çağrıları arasında veri tutabilen değişkenlerdir. Bellek değişkenleri ise, verileri geçici olarak tutabilen değişkenlerdir, örneğin bir fonksiyonun yürütülmesi sırasında. Anlaşılabileceği gibi, genellikle fonksiyon değişkenlerinin çoğu bellek değişkenleridir ve global değişkenlerin tümü depolama değişkenleridir.

Güvenlik açığı, Solidity'nin depolama değişken türlerini ve karmaşık veri yapılarını depolama biçiminden kaynaklanır, 32 bayt sıralı yuvalarda saklanırlar. Yapılar (Struct) gibi yerel olarak karmaşık yapılar tanımlandıklarında, depolama değişkenleri olarak da değerlendirilir. Böylece diğer depolama değişkenleriyle aynı yuvalara yerleştirilirler. Bu, tanımlanmamış bir değişkenin başka tamamen normal depolama bir değişkenine yerleştirilebileceği ve üzerinde yapılacak herhangi bir değişikliğin normal değişkeni etkileyeceği anlamına gelir [37].

3.1.10.1. Bilinen ataklar

Bu güvenlik açığının gerçek dünyadaki uygulaması diğerlerinden biraz farklıdır çünkü söz konusu kontrat aslında insanlardan bir miktar Ether almayı amaçlayan bir bal küpüdür (honeypot). İlk bal küpü (honeypot),

0x741F1923974464eFd0Aa70e77800BA5d9ed18902 adresinden erişilebilen OpenAddressLottery'dir [43].

Bu kontrat, bir yapı (struct) olan *s* adlı bir değişken içerir ve tahmin edilebileceği gibi değişkene bir değer atanmamıştır. Bu, sonunda saldırganların kontrattaki gizli numarayı değiştirmesine yol açacaktır [44].

3.1.10.2. Öneriler

Solidity 0.5.0'dan itibaren Solidity derleyicileri bu güvenlik açığı hakkında bir uyarı yayınlar ve Solidity 0.10'dan itibaren derleyiciler başlatılmamış bir yerel değişken varsa bir hata verir [45]. Ancak buna ek olarak tüm değişkenleri tanımlamak esastır. Ayrıca, depolama ve bellek anahtar sözcüklerinin açıkça kullanılması, bu güvenlik açığından kaçınmak için iyi bir uygulamadır.

3.2 Sonuç

Daha önce de bahsedildiği gibi dağıtım süreci sonucunda 1 milyon sözleşme içeren bir veri seti üzerinde analiz yapılmıştır. Bu sözleşmelerde 127,644 güvenlik açığı tespit edildi. Güvenlik açıklarının dağılımı çizelge 3.1'deki tabloda verilmiştir:

Çizelge 3.1 : 2020 akıllı kontrat zafiyet dağılımı.

İsim	Sayı
Reentrancy	35.209
*Block.timesta*mp Değişkeni	19.527
Katı Eşitliklerin/ Assert Kullanımı	18.523
Rastgele Hedeflere Ether Gönderme	17.910
Kilitli Ether	11.327
Eksik Sıfır Adres (0x0) Doğrulaması	11.283
Yerel Değişken Gölgeleme	3094
Kullanılmayan CALL Çıktıları	1830

Zayıf Rastgele Sayı Üretimi (PRNG)	1104
Tanımlanmamış Yerel Değişkenler	909
Diğer	6928

Güvenlik açıklarının %27,57'sini oluşturan yeniden giriş güvenlik açıkları en çok karşılaşılan güvenlik açıklarıydı. Bunu %15,29 ile blok zaman damgası kullanımından kaynaklanan zafiyetler takip ediyor. Bir başka ilginç sonuç da 2020'deki en büyük 10 güvenlik açığının %60'ının altyapı veya mantık hatalarından ziyade kullanıcı ihmalinden kaynaklanmasıdır. Bunun nedenleri, Solidity'nin rastgele sayı üretimi gibi Ethereum altyapısındaki kısıtlamaları aşmaya yönelik çözümlerinin tam olarak olgunlaşmamış olması ve Ethereum geliştirici ortamının güvenli kod geliştirme tekniklerine aşina olmaması olabilir.

4. ÖNERİLEN METODOLOJİ

Tez çalışmasının bu bölümünde, akıllı kontratların denetimi için kapsamlı bir metodoloji önerilmeye çalışılacaktır. Bu metodolojide önerilen basamakların sırası uygulamayı yapan kişinin tercihine göre çeşitli değişiklikler gösterebilir. Buna örnek olarak kimi uygulamacılar, kontratı manuel analiz etmeden direkt olarak araç kullanarak analiz etmeyi tercih edebilirler, bu sayede kontratın yapısı ile ilgili ön bilgi sahibi olmuş olurlar.

4.1 Bilgi Edinme

Bir denetim sırasında, süreci ilerletecek olan kişiye genellikle şu materyaller teslim edilir.

- Dokümantasyon
    - Proje dokümanı (Whitepaper),
    - Github *ReadMe* dosyası,
    - Doküman kütüphanesi linki
- Test edilmesi beklenilen kaynak kod (Genellikle sabitlenmiş bir Github Repository'si)
    - Projenin kaynak kodu ,
    - Projenin kaynak kodlarının testleri için yazılmış kodlar

Bilgi edinme aşamasında öncelikle yapılması gereken şey projenin ne yapmayı amaçladığını, işletme mantığını, dizayn ve mimari yapısını anlamak olmalıdır. Bu genellikle sağlanan dokümantasyonların detaylı bir biçimde incelenmesi ile mümkün olur. Ancak dokümanın eksik kaldığı görülen yerlerde proje yetkilileri ile görüşmeler yapmak da etkili bir çözümdür. Bu doküman inceleme süreci sayesinde denetim işlemini gerçekleştiren kişi proje hakkındaki aşağıdaki bilgileri öğrenebilir:

- Projenin çeşitli bileşenlerinin hangi amaçlara hizmet ettiği,
- Geliştirici ekibin varsayımları, kullandıkları formüller, amaçları,
- Aynı zamanda bileşenlerin bu amaç ve varsayımları nasıl gerçekleştirdiği/ uyguladığı,

- Projede kullanılan varlıklar;
    - İç ve dış kütüphaneler, sınıflar, kullanıldıysa assembly kodları, değişkenler, objeler, fonksiyonlar
- Bu varlıkların birbirleri ile nasıl etkileştiği,
- Eğer proje rol bazlı erişim tanımlamış ise, buradaki rollerin neler olduğu ve ne kadar erişim izinlerinin olduğu

Bu bilgiler ışığında projedeki potansiyel zayıflıklar hakkında varsayımlarda bulunmak mümkündür. Bu varsayımlar projenin kullandığı dile ait zayıflıklar ile sınırlı kalmayıp, mantıksal zafiyetlere yol açabilecek uygulamaları ve geliştirici ekibin gözden kaçırdığı noktaları da içermektedir.

Burada değinilmesi gereken iki önemli nokta bulunmaktadır. Bunlardan ilki bu işlemin vakit alan bir süreç olmasıdır. Diğeri ise bu kadar detaylı dokümanlarla gerçek hayatta karşılaşmak son derece nadirdir. Genellikle belge olarak bu projelerde bir Github *ReadMe* dosyası veya kabaca işletme mantığını içeren internet sitesi verilir. Haliyle bu gibi durumlarda yukarıdaki bilgilere ulaşmak mümkün olmaz, bu sebeple kaynak kodun incelenmesi gerekir. Bu da tahmin edilebileceği gibi zaten uzun olan süreci daha da uzatır. Kaynak kodun incelenmesi ile ilgili detaylı bilgiler bir sonraki başlıkta anlatılacaktır.

Bilgi edinme aşamasında süreci hızlandırmak için çeşitli araçlardan faydalanmak da mümkündür. Bu araçlar genellikle projedeki kontratları grafikler haline getirerek, varlıklar ve roller arasındaki etkileşimlerin daha kolay bir şekilde anlaşılmasını sağlarlar. Ancak unutulmamalıdır ki bu araçlar yukarıda bahsedilen bilgilerin edinilmesine yardımcı olmak amaçlıdır, tek başına bu araçların kullanılması yeterli değildir. Bu araçlara örnek vermek gerekirse:

- **solgraph [46]:** Bir Solidity sözleşmesinin işlev kontrol akışını görselleştiren ve olası güvenlik açıklarını vurgulayan bir DOT grafiği oluşturur**.**
- **Slither [23]:** Aslında bir statik zafiyet analiz aracıdır. Ancak *printer* isimli özelliği sayesinde aşağıdaki gibi özetler sunabilmektedir:
    - human-summary: Sözleşmelerin insan tarafından okunabilir bir özetini yazdırır.

- inheritance-graph: Her sözleşmenin inheritance grafiğini bir nokta dosyasına aktarır.
- contract-summary: Sözleşmelerin bir özetini yazdırır
- call-graph: Sözleşmelerin çağrı grafiğini bir nokta dosyasına aktarır.
- cfg: Her işlevin CFG'sini dışa aktarır.
- function-summary: Fonksiyonların bir özetini yazdırır.
- vars-and-auth: Durum değişkenlerinin ve fonksiyonların yetkilendirmesini yazdırın.

4.2 Manuel Analiz

Manuel analizi iki alt başlıkta incelemek uygundur, bunlar; manuel inceleme ve manuel test olarak isimlendirilebilir. Manuel analiz araç analizine göre daha yavaş ve efor isteyen bir süreç olmasına rağmen, bir önceki aşamada bahsedilen mantıksal hataları bulma ve kodun hedefinin dışında bir amaca hizmet ettiğini anlamak gibi konularda daha etkilidir. Yani kısacası işletme mantığından veya modellemeden kaynaklanan zafiyetleri bulmak için daha çok tercih edilmesi gereken bir yoldur. Bunun en önemli nedenlerinden biri, Ethereum ekosisteminde her projenin kendine özgü bir yapısı olmasından kaynaklanır. Bu, Ethereum ortamındaki tüm projelerin benzersiz olduğu anlamına gelmez. Ethereum ekosisteminde birbiri ile benzer işleri yapan, aynı alanda çalışan pek çok projeye rastlamak mümkündür. Buradaki fark, bu projelerin kendi ele aldıkları sorunları biraz farklı bir çözümle aşmaya çalışması olarak basite indirgenebilir. İşte bu benzersiz yapı sebebi ile her projenin işletme mantığı ve kullandığı modeller birbirlerinden farklı oluyor. Bunlarda ortaya çıkan zafiyetleri bulabilmek için otomatik analiz yöntemi kullanmak hem çok efektif sonuç vermemektedir (yanlış pozitif ve yanlış negatif bağlamında) hem de bu araçların bu amaçla kullanımı tam anlamıyla otomatik bir süreç olmayacaktır. Çünkü bu analiz yöntemlerinin tespit etmesi gereken koşullar için deney durumlarının özel olarak yazılması gerekir. Bu sebeplerden dolayı manuel analiz bir denetim sürecine mutlaka dahil edilmelidir.

4.2.1. Manuel inceleme

Manuel inceleme projenin kaynak kodunu statik bir biçimde incelemek olarak tanımlanabilir. Bu işlem bir önceki aşamada de bahsedildiği gibi dokümantasyonun yetersiz olduğu durumlarda proje hakkında bilgi edinme amacıyla yapılabildiği gibi denetim işleminin bir parçası olarak da yapılabilir. Bu süreci yürüten kişinin tasarrufundadır. Eğer proje dokümanları daha önce de bahsedildiği gibi eksik veya yeterli bilgileri içermiyorsa bu aşamada bir önceki başlıkta bahsedilen bulgular elde edilebilir ve potansiyel zafiyetler için varsayımlar yapılabilir. Bunun yanı sıra yazılım için tercih edilen dile, işletme mantığına, kullanılan kütüphanelere bağlı olan zafiyetleri bu aşamada tespit etmek mümkündür.

Bir projede manuel inceleme yaparken dikkat edilmesi gereken noktalardan en önemli birkaç tanesi şu şekildedir

4.2.1.1. Anlık krediler

Bu çalışmanın önceki bölümlerinde anlık kredilerin ne olduğu ve nasıl çalıştığı ile ilgili detaylı bilgi bulmak mümkündür. Yine de kısa bir hatırlatma yapmak gerekirse anlık krediler, yüksek miktardaki fonları gerektiren işlemler için sağlanan teminatsız kısa süreli kredilerdir. Bu krediler belirtilen süre içerisinde ödünç alınan miktarın geri verilmesi şartı ile dağıtılır. Şayet bu geri verme gerçekleşmezse yapılan işlem geri çevrilir. Bu Ethereum'un sağladığı yararlı bir özelliktir, ancak bu özellik pek çok saldırıda da kullanılmaktadır. Her ne kadar şüpheli işlemleri tespit etmeye yarayan pek çok izleme aracı ve yöntemi bulunsa da akıllı kontratların yapısı gereği bu tarz anomalileri mümkünse kontrat blokzincire yüklenmeden önce tespit etmek daha iyi olacaktır.

Anlık kredilerin kullanımında dikkat edilmesi gereken noktalara değinmeden önce anlık kredilerin Ethereum ekosisteminin sunmuş olduğu yararlı ve yasal bir özellik olduğunu tekrar hatırlatmakta fayda var. Anlık kredilerin kullanıldığı projelerde denetim işlemi gerçekleştiren kişilerin dikkat etmesi gereken noktalar şunlardır:

- Sistemdeki fonların veya token'ların azlığına dayanan varsayımların olup olmadığına dikkat edilmeli. Bu tarz varsayımların tespiti dokümantasyonlardan anlaşılabileceği gibi genellikle token veya fonlar için swap işlemleri gerçekleştiren fonksiyonların girdileri manipüle edildiğinde de görülebilir.

- Aynı şekilde bu tarz projelerin kullanıcıların alabileceği anlık krediler için bir üst değer belirlemesi de saldırıları önlemeye yardımcı olabilecek bir davranıştır. Bu sayede özellikle arbitraj kullanımına izin veren projelerde saldırganın çok büyük değerde anlık krediler çekerek yapacağı saldırılar önlenebilir. Ancak bu yöntem genellikle çok tercih edilmez, çünkü bu tarz bir sınırlama projenin kullanımını da sınırlayabilir.
- Anlık kredilerin kullanımında dikkat edilmesi gereken en önemli nokta ise anlık kredi kullanımı içeren isteklerin tek bir işlemde blokta taşınmasının önüne geçmek olacaktır. Bu sayede yukarıda anlatılan önlemler alınmamış olsa bile saldırganların anlık krediyi kullanarak projenin kontrollerini geçip bulduğu zafiyetleri tetiklemesi engellenebilir.

4.2.1.2. Anlık mint

Anlık mint işlemi, bir kullanıcının kendi hesabına istediği sayıda token yaratmasına olanak verir, buradaki önemli nokta işlem bitmeden önce basılan token kadar token yakılmak (kullanılmak) zorundadır [47]. Bu tarz anlık mint işlemleri kullanan kontratlarda çeşitli kontrollerin olması beklenir. Aksi takdirde oluşabilecek dalgalanmalara karşı proje korunaksız kalır ve saldırganlar bunu sömürebilir. Şayet böyle kontrollerin olmadığı durumlarda işletme mantığının kullandığı ekonomik modelin anlık mint işlemlerinin kötüye kullanılmasını engelleme şansı vardır. Ancak bu çok kapsamlı ve iyi düşünülmüş bir ekonomik model kullanılması anlamına gelir ve çoğu zaman bu kadar uç noktaların testleri yapılamaz. Aynı şekilde bu tarz zafiyetleri tespit etmek de çok zordur. Çünkü proje ile aynı alanda derin ekonomik bilgi gerektirir. Bu sebeple projeye kontrollerin uygulanması hem proje sahipleri hem de denetim işlemini gerçekleştiren taraflar için daha kolay ve efektif bir çözüm sunar.

4.2.1.3. ERC Token'larının kullanımı

ERC yapılarını Ethereum ekosistemindeki token'lar için birer API standardı olarak düşünmek makul olacaktır. Her bir ERC standardı farklı bir amaca hizmet etmektedir. Bunlara örnek vermek gerekirse:

- ERC20 [48]: Değiştirilebilir (Fungible) Token'lar için yaratılmış bir ortak standarttır. Bu sayede her token'ı başa bir token'a çevirmek mümkün olur.

- ERC771 [49]: ERC20'nin kapsayamadığı durumları kapsamak için yaratılmış daha kompleks bir standarttır. Değiştirilemez (Non-Fungible) Token'lar (NFT) için aidiyet temsil etmeyi amaçlar.
- ERC884 [50]: ERC884 token'ları David Sag'ın standardına göre bir Delaware şirketindeki belirli bir payı tanımlar. Bu standart hisse işlemleri için tasarlanmıştır ve token sahibi akıllı kontratta yerleşik bir yöntem olan beyaz listeye alınmalıdır. ERC-884 ihraççıları ise menkul kıymetler mevzuatına uymak için zincir dışı özel bir veri tabanı oluşturmalıdır.
- ERC621 [51]: ERC621, ERC20 standardının bir uzantısıdır. Dolaşımdaki toplam token miktarını artırmak ve azaltmak için iki fonksiyon ekler.
- ERC721 [52]: ERC-721, Ethereum blokzincirde değiştirilemez veya benzersiz token'ların nasıl oluşturulacağını açıklayan ücretsiz, açık bir standarttır. Çoğu token değiştirilebilir olsa da (her token diğer token'larla aynıdır), ERC-721 token'larının hepsi benzersizdir.

ERC token'larının kullanıldığı projelerde dikkat edilmesi gereken bazı noktalar şunlardır:

- ERC token'ları ile etkileşime giren fonksiyonların sonuç değerleri kontrol edilmelidir. Buna örnek olarak ERC20 token'larında kullanılan *transfer* fonksiyonları örnek verilebilir. Bu fonksiyonlar çağrıldıklarında *boolean* değerler dönmektedirler. Bu tarz fonksiyonları çağıran projelerde fonksiyon çıktıları kontrol edilmelidir, aksi takdirde yapılan işlem başarısız olabilir. Burada yapılan işlemin başarısız olmasından daha ciddi bir sonuç da gözlemlemek mümkündür. Örneğin ERC20'nin belli uygulamalarında *transfer* ve *transferFrom* fonksiyonları sonuç olarak sadece *boolean* değerler döner. Normalde yapmaları gereken davranış ise bir değer dönmekten ziyade herhangi bir terslikte işlemi geri çevirmektir. Sadece *boolean* döndükleri için işlem bu çıktılar kontrol edilmezse yanlış hali ile devam edebilir.
- ERC token'larında ölçeklendirme işlemleri için ondalık sayılar kullanılır. Bu sayıların çeşitli standartlar için kullanılan programlama dillerinde varsayılan değişken türleri bulunmaktadır. Bunlara örnek olarak ERC20 token'larda ondalık sayıların Solidity'de *uint8* türünde saklanması örnek gösterilebilir [53]. Şayet böyle bir varsayımın olduğu durumlarda, varsayılan türün dışında

bir değişken türü çıktı olarak verilirse bu çeşitli öngörülemez davranışlara yol açabilir.

- Erc20 token'larına özgü bir yarış kondisyonu bulunmaktadır. Bu standartta bulunan *approve* ve *transferFrom* metotları bu tarz bir zafiyete yol açmaktadır. Burada dikkat edilmesi gereken nokta, *approve* metodu ile herhangi bir kontrol olmadan mevcut ödenek değeri değiştirilebilir. Daha sonra değişen ödenek limitleri içinde *transferFrom* ile fon transferi sağlanır [54].
- ERC token'larının belirli uygulamaları reentrancy zafiyetine yol açabilir. Dışarı çağrı yapmaya izin veren yapıları kullanan uygulamalarda bu risk görülür. Bunun en bilinen örneği ERC777 standardındaki çengel (hook) yapısıdır [53].
- Erc621 token'larında rol bazlı yetkilendirme vardır. Total arz, yetkisi olan tüm kullanıcılar tarafından değiştirilebilir [55].
- Hiçbir kullanıcı token miktarının büyük çoğunluğuna tek başına sahip olmamalıdır. Bu durum genellikle proje dizaynında gözden kaçabilecek bir detaydır. Ancak bir kullanıcının projedeki token miktarının büyük bir kısmına sahip olması, projenin kontrolünü kullanıcıya teslim etmeye kadar gidebilir.

Kısacası ERC token'larını kullanan projelerde ilgili fonksiyonların girdiler, çıktıları, hata halletme şekilleri, erişim yetkileri dikkatli bir biçimde incelenmelidir.

4.2.1.4. Erişim denetimi ve rol bazlı erişim

Çoklu rol tanımının olduğu projelerde, tanımlanan rollerin projedeki objelere, fonksiyonlara müdahale yetkileri, tanımları, tanımlı güvenilirlikleri dikkatle incelenmelidir. Bu incelemelerde eğer varsa güvenilen rollerin neler olduğu ve hangi varsayımlar ile projenin hayata geçirildiği incelenmelidir. Normalde blokzincir teknolojisi güvensiz bir yapıya sahiptir. Burada güvensiz yapı ile anlatılmak istenilen, Bizantin Tehdit Modeli'ne uygun şekilde, projeye erişme yetkisi olan tüm kullanıcılara potansiyel saldırgan gözüyle bakılmasıdır. Ancak bazı projelerde tanımlanmış güvenilmiş roller bulmak mümkündür, hatta belirli prosedürler bu güvenilir rollerden faydalanmayı amaçlar. Bu prosedürlerin en bilinen örneği korunaklı piyasaya sürmedir (guarded launch) [76]. Bu prosedür güvenilir roller ile başlayıp her yinelemede daha az merkezi olmaya giden bir yapıdır. Tanımlı rollerin

bulunduğu projelerde dokümantasyon yardımı ile roller ve rollerin kapsamları belirlenebilse de implementasyonda bu rolleri tekrar kontrol etmek gerekir.

Tanımlı rollere ve rollerin yetkilerine dikkat etmek kadar, tanımlı roller arasında geçiş yapılmasını sağlayan fonksiyon ve yapılara dikkat etmek de önemlidir. Şayet bu durumlarda kontratın kilitlenmesi, erişim dışı kalması veya rollerin ele geçirilmesi gibi senaryolar ortaya çıkabilir. Bu tarz durumlarda, özellikle rollerin yetki kapsamları genişledikçe, ekstra kontrollerin uygulanıp uygulanmadığına bakılmalıdır.

4.1.2.5. Ether, gaz ve veri akışı idaresi

Proje içerisinde ve proje dışına gerçekleşen veri akışlarının takip edilmesi ve doğrulanması gerekir. Bu sayede kullanıcı kaynaklı veya altyapısal pek çok zafiyetin önüne geçmek mümkündür. Aynı zamanda akışın takibi kontratlardaki depolama (storage) ve bellek (memory) alanlarındaki değişiklikleri de takip etme olanağı sunar. Basitçe proje içerisindeki veri akışını takip etmek için projedeki kontratlar arası işlemler yapmaya olanak sağlayan fonksiyonların girdi ve çıktılarına dikkat etmek yeterli olacaktır. Aynı şekilde proje dışı ile ilişkili veri akışlarını incelemek için proje dışı ile iletişim kuran fonksiyonların girdi ve çıktıları incelenmelidir.

Bu akışların incelenmesinin yanı sıra taşınan verinin doğrulanması da önemli bir mevzudur. Bir projede taşınan verinin doğrulanması ile uğraşan mekanizmaların dikkatle incelenmesi gerekir. Bu mekanizmalarda yapılan varsayımlarda, gelen verinin doğrulanması gerektiği çoğunlukla atlanılabileceğinden veya eksikler barındıracağından doğrulanmamış verilerin akışına sıkça rastlanır. Ancak unutulmamalıdır ki, sistemlerde genellikle doğrulanmamış veri akışı beklenmeyen davranışlara yol açar.

Akıllı kontratlarda verinin akışı kadar Ether idaresi de kritik bir noktadır. Burada idare ile anlatılmak istenilen Ether'in transferi ile ilgili işlemlerin yanı sıra Ether'in kontrat içerisinde de yönetilmesidir. Ether'in proje içerisinde muhasebesini yapan birimler, roller, izinler, durum değişiklikleri, girdiler azami özenle incelenmelidir.

Ether idaresinin yanı sıra Gaz idaresi de bir proje için çok önemlidir. Bir proje her ne kadar sağlam, yenilikçi olursa olsun Gaz idaresi doğru yapılmadığında veya manipüle edilebilir olduğunda, projenin işlem sıklığı düşmeye başlayacaktır. Bu da aslında projenin başlarda yeni kullanıcı çekememesi ve neticede kullanıcı kaybı

yaşamasına yol açacaktır. Hatırlanacağı gibi Gaz Ethereum ekosisteminde gerçekleşen işlem maliyetlerini takip etmek ve düzenlemek için kullanılan bir yapıdır. Projelerde Gaz ile ilgili varsayımların yapıldığı çağrılar, döngüler gibi yerlere özellikle dikkat edilmelidir.

4.2.1.6. Harici bağımlılıklar

Günümüzde herhangi bir yazılım projesinin dış bağımlılığı olmaması beklenemez. Bu Ethereum ekosistemi için de geçerlidir. Ethereum ekosistemindeki projelerin dışarı bağımlılıkları şu şekilde özetlenebilir:

- *Kahin*
- Blok değişkenleri
- Kütüphaneler
- Token'lar
- Başka kontratlar

Harici bileşenler ile etkileşimler geliştiricileri bu bileşenlere güvenmeye, bu bileşenlerden gelebilecek potansiyel sonuçlar için belirli varsayımlar yapmaya iter. Ancak harici bileşenlere güvenmek veya yapılan varsayımların yeteri kadar kapsamlı olduğunu düşünmek güvenlik zafiyetlerine yol açar. Projelerde harici etkileşimlerin olduğu kısımlara özellikle dikkat edilmelidir. Bu tarz zafiyetlere örnek olarak bu çalışmada bahsedilen SpankChain ve StableMagnet olayları verilebilir.

Bu başlığın kalanında yukarıda listelenmiş dikkat edilmesi gereken bileşenlerden bahsedilecektir. Token'lar için bu başlığın önceki bölümleri incelenebilir. Benzer şekilde Blok değişkenleri için ise 3.1.2 başlığına bakılabilir. Ethereum projelerinde en sık kullanılan dış bileşenlerden biri kütüphanelerdir. Ethereum'un açık kaynaklı yapısı sayesinde pek çok kütüphanenin kaynak koduna erişerek bunları kötücül amaçlar için manipüle ederek yeni kütüphaneler yayınlamak mümkündür. Bu yüzden geliştiricilerin kullandıkları kütüphanelerin doğrulanmış veya esas kaynaktan olduğuna emin olmaları gerekir. Bunun yanı sıra erişilen harici kontratlarda veya kullanılan kütüphanelerde çeşitli zafiyetler bulunabilir ve bu zafiyetler projenin hedef olmasına yol açabilir. Tabii harici bileşenler genelde denetim kapsamının dışında yer alacağından bunlardaki zafiyetleri bulmaya çalışmak manasız ve zaman açısından pahalı olacaktır. Bunun yerine geliştiricileri kullandıkları harici bileşenlerin

kaynaklarını kontrol etmeye ve harici bileşenlerin yapılmış denetim raporlarını incelemeye teşvik etmek gerekir. Denetçiler ise bu bileşenlerin herhangi bir olayda hedef alınıp alınmadığının kontrol edilmesini sağlamakla yükümlülerdir. Benzer şekilde kullanılan kütüphanelerin kaynaklarının da kontrol edilmesi de yapılması gereken uyarılardandır.

4.2.1.7. İşletme mantığı

Bir projenin işletme mantığı, hangi alanda çalışacağından, ekonomik modeline kadar genel olarak projenin sunmayı amaçladığı çözümün nasıl olacağının belirlendiği yerdir. İşletme mantığındaki gözden kaçırılan noktalar, eksik varsayımlar projeler için çok tehlikeli olabilir ve ön görülmeyen davranışlara yol açabilir. Saldırganlar bu eksiklikleri manipüle ederek ciddi zararlar verebilirler. Bu başlıkta bahsedilen eksik varsayımlar projede tanımlanan rollerin yetkileri, projenin belirli şartlar altında nasıl tepki vereceği gibi konularda olabilir. Genellikle belirli şartlar altında projenin sergilemesi gereken davranışlar tanımlansa da bu şartların dışına olabilecekler eksik bir şekilde tanımlanmaktadır. Benzer şekilde kullanılan ekonomik modelin çok iyi bir biçimde tasarlanmış olması gerekir. Ekonomik modelin üzerinde yapılan çalışmalarda eğer modelin kendisinde bir açık bulunursa, kodlama ve dokümantasyon ne kadar iyi ve dikkatli bir şekilde yapılırsa yapılsın proje yine de güvende değildir. Buradaki önemli nokta denetçinin modelin tasarlandığı alan yeterli bilgi birikimine sahip olup olmadığıdır. Çoğu zaman denetçilerin projelerin ekonomik modelleri ile ilgili basit ölçüde, anlayacak kadar, bilgi birikimleri olduğundan ekonomik modelin üzerinde çalışma yapılması ve ekonomik modelin mantığında açık bulmak oldukça zordur. Ancak bu bir denetim prosedüründe ekonomik model ile ilgili belirli bir anlayışa sahip bir kişinin projede modelin yapısına aykırı davranışları tespit edebilmesi ile karıştırılmamalıdır.

Bu anlatılanların dışında üçüncü bölümde yer alan zafiyetlere yol açabilecek bulguları da bu aşamada tespit etmek mümkündür. Ancak burada unutulmaması gereken nokta Ethereum ekosisteminde ortaya çıkabilecek zafiyetler kesinlikle bunlar ile sınırlı değildir. Bilinen zafiyetlerin üzerine her geçen gün bir yenisi eklenmektedir.

4.2.2. Manuel test

Manuel test yönteminde projeler bir test ortamına yüklenir ve burada dinamik bir şekilde incelenir. Burada dinamik derken anlatılmak istenilen, projenin hali hazırda bulunan testlerine bakmanın yanı sıra yeni test senaryoları denemek olarak genişletilebilir. Bu sayede denetim işlemini gerçekleştiren partiler projenin olgunluğu, projenin sahip olduğu çeşitli varsayımlar ve uç noktalar hakkında bilgi edinebilir. Bu tarz test işlemleri için piyasada hazır bulunan Truffle [77], Hardhat [78], Brownie [79] gibi geliştirme ortamlarından faydalanılabilir. Ancak uç noktaların tespiti için manuel test yöntemi kullanmak oldukça eforlu bir iştir. Projenin hali hazırda bulunan testlerinden faydalanarak uç noktaları tespit etmek kolay olsa da projenin testlerinde yer almayan kısımları için bu tespiti manuel olarak yapmak zaman ve efor bakımından çok maliyetlidir. Bu yüzden uç noktaların tespiti için manuel testin yanına araçlar da eklenmelidir.

Bunların yanı sıra belirli skorun üzerindeki zafiyetler, proje yetkililerine kanıt ile daha iyi anlatılabileceğine inanılan bulgular ve manuel inceleme ve statik analiz sırasında tespit edilen ancak hatalı pozitif olmasından şüphelenilen bulgular için ispat niteliğinde sömürü kodları ve bu kodların test senaryolarını sunmak iyi bir uygulamadır. Bu sayede bulgulardaki hata oranı azaltılabilir ve bulgular daha gerçekçi ve ilişki kurulabilir hale gelir. Bu nedenle belirli yerlerde kavramsal kanıtlama kullanmak denetim için iyi bir uygulamadır.

4.3. Araç analizi

Akıllı kontratlarda zafiyet analizi yapmak için kullanılan araçların faydalandığı üç ana teknik vardır. Bunlar sırası ile statik analiz, dinamik analiz ve formül ile doğrulama yöntemleridir. Bu çalışmanın takip eden kısmında bu teknikler ve bu teknikleri uygulamak için kullanılan yöntem ve araçlardan bahsedilecektir. Bölüm boyunca teknikler, aşağıdaki maddeler göz önünde bulundurularak değerlendirilecek ve hangi tekniğin bu çalışmada önerilen metodoloji için en uygun olan olduğu belirlenmeye çalışılacaktır:

- M1: Tekniğin Sonuç Efektifliği
- M2: Teknik Kısıtlar

NOT: Bölümde yapılması planlanan karşılaştırma analiz teknikleri arasında olup, bahsedilecek araçlar arasında kesinlikle bir karşılaştırma yapılmayacaktır.

4.3.1. Araç ile statik analiz

Akıllı kontratlarda statik analiz, bir kontratın kaynak veya bayt kodu üzerinde yapılan analiz işlemleridir. İsminden de anlaşılabileceği gibi statik analiz, kontrat herhangi bir bloğa yüklenmeden yapılır. Bu tür analizleri kod üzerinde yapılan hata ayıklama (debug) işlemlerine benzetmek doğru olacaktır. Statik analizdeki asıl amaç kontrat kodlanırken yapılan mantık hatalarını ve güvensiz kodlama pratiklerini bulmaktır [2], bu sayede kontrat bir blokta çalışmaya başladığında bunlardan kaynaklanan zafiyetlerin istismarı önlenmiş olur.

Geleneksel statik analiz işleminin işleyiş biçimi aşağıdaki gibidir:

1. Analiz için kaynak kodu veya bayt kodu girdi olarak verilir.
2. Girdi olarak alınan koddan derleyici yardımıyla bir *AST-JSON* dosyası üretilir.
3. Çıkan AST-JSON dosyasından kontratın fonksiyonları ve çıktıları hakkında bilgi edinilir.
4. Zafiyetler için önceden tanımlanmış olan testler bir önceki aşamanın sonucunda oynatılır.
5. Çıkan sonuçlardan zafiyetler tespit edilir.

Statik analiz yapılırken çokça tercih edilen bir başka yöntem de *sembolik yürütmedir.* Sembolik yürütmedeki mantık programın ihtiyaç duyduğu girdiler yerine sembolik değerler kullanarak programın çalışmasını simüle etmek ve bu sayede program çalıştığında ortaya çıkabilecek kısıtlamaları ve dallanmaları toplamaktır. Sembolik yürütme metodu dinamik ve statik olmak üzere ikiye ayrılır. Bu ikisi arasındaki fark, dinamik sembolik yürütmede program gerçek uygun değerler kullanılarak defalarca çalıştırılır ve eğer bir dallanma bulunursa, bulunan dallanmalardan biri seçilip devam edilir. Daha sonra tüm döngü daha önce bulunan bir dallanmadan tekrar başlar. Statik sembolik yürütme de ise böyle bir durum söz konusu değildir [56]. Özetlemek gerekirse sembolik yürütmedeki asıl amaç bir programın işleyiş yollarını ve dallanmalarını keşfetmektir. Bu başta zafiyet keşfi ile çok alakalı bir yöntem gibi

görünmese de unutulmamalıdır ki, zafiyetler de programların aldığı belli girdiler sonucu ortaya çıkan dallanmalardır.

Çoğu açık kaynaklı olmak üzere akıllı kontralarda statik analiz yapmak için geliştirilmiş pek çok araç mevcuttur. Bu araçlar çeşitli geleneksel statik analiz yöntemlerini veya bu yöntemlerin kombinasyonlarını kullanarak kapsamlarını olabildiğince geniş tutmayı amaçlamışlardır. Öne çıkan statik analiz araçlarından bazıları bulunabilir:

- **Slither [23]:** Slither Trail of Bits tarafından geliştirilmiş olan Python bazlı bir statik analiz aracıdır. Analizlerinde zafiyet tespitinin yanı sıra optimizasyon ve kod yapısı ile ilgili bilgiler de verebilme yetilerine de sahiptir. Aynı zamanda sunduğu API ve grafik gösterimler, özel ve otomatik analiz yapabilme gibi özellikler ile kullanım kolaylığı sağlamayı amaçlar. 70'in üzerinde farklı bulgu türüne sahip olan araç, bulgular ile ilgili detaylı bilgi ve bunlardan korunma yolları sunar. Ayrıca Slither analizlerini ortalama olarak 1 saniyenin altında bir sürede tamamlamaktadır.
- **SmartCheck [57]:** Çalışma mantığı ve sonuçlar bakımından Slither'e çok benzer bir yapıya sahiptir ve aynı Slither gibi bulguları hakkında detaylı bilgi verebilmektedir. Javascirpt bazlı yazılmış olan araç aynı zamanda bir online olarak bir web sunucuda da hizmet vermektedir. Bu sayede kurulum ve ortam hazırlama gibi eforlardan kullanıcısını kurtarmayı amaçlar. 4600 valide edilmiş kontrat üzerinde yapılan denemelerde, kontratlarda %99,9 bir güvenlik açığı bulan araç bulduğu açıklardan %63,2'sini ciddi birer zafiyet olarak sınıflandırmıştır [41].
- **Oyente [58]:** Oyente 2016 yılında çıkmış bir araç olup geleneksel statik analiz yöntemleri yerine sembolik yürütmeden faydalanmaktadır. Yaratıcıları tarafından yapılan çalışmalarda 19,366 kontrattın 8,833 tanesinin zafiyetli olduğunu bulmuştur.
- **Securify [59]:** Securify statik sembolik yürütme metodunu benimsemiş açık kaynaklı bir araçtır. 38 ayrı zafiyet modeline sahip araç bunları kullanarak rapor üretir. Python kullanılarak yazılmış olan aracın iki versiyonu bulunmaktadır. Diğer sembolik yürütme araçlarından farklı olarak, Securify bulduğu her dallanma üzerinde ayrı ayrı statik analiz yapar [41].

- **Mythril [60]:** Mythril EVM bayt kodu için statik analiz yapmak üzere tasarlanmış açık kaynaklı bir araçtır. Araç daha sonrasında kaynak kodu desteği de getirmiştir. Python bazlı bir yapıya sahip olan Mythril sembolik yürütme yöntemini benimsemiştir. Çalışma yapısı olarak Oyente ile benzerlik göstermektedir. Zafiyet kapsamı olarak var olan en geniş araçlardan biridir. Ayrıca araç kendisi ile aynı firma tarafından geliştirilen Mythx gibi araçlar ile uyum içerisinde çalışabilmektedir.

4.3.1.1. M1: Sonuç efektifliği

Daha önce de belirtildiği gibi statik analiz işlemi bir nevi kod üzerinde yapılan hata ayıklama gibidir. Bu sebeple statik analiz işlemlerinde daha önce tanımlanmamış bir bulgunun bulunması beklenmez. Ancak tanımlı bulguların bulunması konusunda statik analiz işleminin sadece akıllı kontratların analizinde değil diğer alanlarda da çokça tercih edilen ve iyi sonuç veren bir yöntem olduğunu belirtmek gerekir. Bu başlık altında ele alınan araçlar arasında karşılaştırma yapılmayacağı daha önce de belirtilmiştir. Ancak Sayeed ve ark. [4] ve Praitheeshan ve ark. [2] gibi çalışmalar incelenirse bu araçların kapsamları ve birbirlerine göre performansları hakkında fikir sahibi olunabilir.

Statik analiz yaklaşımının sağladığı başka bir avantaj da analiz süresinin diğer tekniklere göre daha kısa olmasıdır. Bu gerek geleneksel statik analiz gerek de statik sembolik yürütme yöntemlerinin işleyiş biçiminden kaynaklanmaktadır. Buna bir örnek olarak sembolik yürütmenin çalışma mantığı verilebilir.

Sembolik yürütme çalışma prensibi sayesinde bir programın farklı girdiler altında yarattığı birden fazla dallanmayı keşfedebilir [61]. Bu hem kapsamlı hem de çabuk bir şekilde analizlerin sonuçlanmasını sağlar.

Statik analiz kapsamlı ve çabuk sonuç verebilmesine karşın, sonuçlarındaki hatalı pozitif ve hatalı negatif değerlerinden dolayı sıkıntı yaşamaktadır. Bakıldığında akıllı kontratların analizinde hatalı pozitif sonuç verilmesi o kadar da büyük bir sorun yaratmaktadır. Bu zafiyet olmayan bir kod parçasında zafiyet varmış gibi bir bulguya yol açar, ki böyle bir bulgu en fazla zafiyet analizini yürüten kişinin kod parçasında zafiyet olmadığını anlaması veya bulgunun yanlış olduğunu yaptığı denemeler sonucunda fark etmesi ile giderilebilir. Kısacası bu analizciyi bir tavşan deliğine itmekten daha büyük bir sıkıntıya yol açmaz. Ancak hatalı negatif sonuçların çıkması

büyük sorunlara yol açabilir. Bir analizde hatalı negatif çıkması demek var olan bir zafiyetin bulunamaması anlamına gelir. Ghaleb ve Pattabiraman [71] yaptıkları çalışmada statik analiz araçlarında ortaya bu tarz hatalı negatiflerin para kaybına neden olduğunun altını çizmişlerdir. Aslında araçların kâğıt üzerinde pek çok sorunu bulabilecek şekilde tasarlanmasına rağmen teknik kısıtlamalar nedeni ile hatalı negatiflerin ortaya çıktığını belirtmişlerdir. Bu görüşü kanıtlayan çalışmalara Sayeed ve ark. [4] ve Praitheeshan ve ark. [2] çalışmaları örnek olarak verilebilir.

4.3.1.2. M2: Teknik kısıtlar

Bir üst maddede de belirtildiği gibi statik analiz işlemlerinde daha önceden tanımlanmamış bir zafiyetin tespit edilmesi beklenen bir davranış değildir. Bu durum statik analiz yöntemlerinin tercih ettiği yaklaşımdan kaynaklanmaktadır. Bu yaklaşım daha önce de değinildiği gibi kontratın üzerinde bulgular için önceden tanımlanmış testlerin yürütülmesi olarak özetlenebilir. Bu durum akıllı kontrat güvenliği gibi yeni gelişmekte olan bir alanda pek çok zafiyetin hemen tespit edilememesi anlamına gelir. Statik analiz yöntemlerinin yeni saldırılara karşı koruma sağlamaması verdiği hatalı negatif sonuçlarla beraber bu tekniğin en büyük dezavantajlarını oluşturur.

Sembolik yürütme analiz yaparken şık bir çalışma metodolojisini benimser. Bu sayede sistematik bir biçimde bir programda oluşan tüm dallanmaları test edebilir. Ancak sembolik yürütme metodunun da karşılaştığı belli zorluklar vardır. Baldoni ve ark. [71] yaptıkları çalışmada aşağıdaki durumların sembolik yürütme için sorun yaratabileceğine ve analiz sonucunda hatalı negatifler çıkarmak gibi sonuçları olabileceğini belirtmişlerdir:

- Sembolik yürütme sırasında array, pointer veya benzeri kompleks veri objelerinin işlenmesinin yan etkileri olabilir.
- Sembolik yürütme sırasında kütüphanelere veya sistem kodlarına yapılan çağrıların, program yığını üzerinde programın düzgün çalışmasını etkileyen yan etkileri olabilir.
- Sembolik yürütme sırasında verilen girdiler programdaki döngüleri veya koşullu dallanmaların patlamasına neden olabilir.
- Sembolik yürütme sırasında lineer olmayan aritmetik işlemler işlemin efektifliğine ciddi darbe vurabilir.

4.3.2. Araç ile dinamik analiz

İsminden de anlaşılabileceği gibi dinamik analiz akıllı kontratları çalıştırarak veya canlı bir ortamda inceler. Bu sebeple dinamik analiz işlemlerinde genellikle kaynak koda veya bayt koduna ihtiyaç duyulmaz. Praitheeshan ve ark [2] bu analiz şeklini gerçekten bir kontrata saldıran bir saldırgana benzetmişlerdir. Dinamik analiz benimsediği bu çalışma yöntemi ile kontratın etkileşim halindeki davranışlarını da gözlemleme şansı sunar.

Sembolik yürütme aynı statik analizde olduğu gibi dinamik analiz tekniğinde de tercih edilen bir yöntemdir. Daha önce de bahsedildiği gibi iki tür statik yürütme bulunmaktadır, burada kullanılacak olan ise dinamik sembolik yürütmedir. Dinamik sembolik yürütme statik olan karşılığından farklı olarak gerçek değerler ve kısıtları kullanarak program akışını yönlendirir. Yeni bir dal ile karşılaşıldığında, bu dalın kısıtları çözülerek bu dal için bir girdi üretilir ve işlem

Programın akışı bitene kadar aynı şekilde devam eder. Dinamik statik analiz dinamik ikili yürütme, sanallaştırma ve hata ayıklama teknolojilerinden yararlanır. İşleyiş biçimi göz önüne alındığında statik karşılığına göre daha uzun sürede sonuç verir [56].

Çalışma zamanı doğrulaması dinamik analiz için kullanılan bir başka yöntemdir. Çalışma zamanı doğrulaması iki farklı şekilde çalışmaktadır. Bunlardan ilki ve en yaygın olanı bir kontratın davranışlarını ağ üzerinde izlemek ve zararlı olduğu düşünülen işlemleri tespit etmek ve reddetmektir. Bu çalışma mantığı bir nevi akıllı kontratlar için yapılmış bir IDS/IPS gibi davranmaktadır. İkinci çalışma şekli de kontratlara koruyucu kodlar yerleştirmeyi amaçlar.

Fuzzing dinamik analiz yöntemleri içinde belki de en çok tercih edilen yöntemdir. Normal bir programa yapılan fuzzing işleminde program rastgele girdiler verilerek çalıştırılır, bu sayede program hakkında bilgi edinilmek amaçlanır. Akıllı kontratlar için yapılan fuzzing de aynı mantık ile ilerler. fuzzing yöntemi genel olarak şu şekilde işlemektedir:

1. Akıllı kontratın ABI'ı veya bayt kodu incelenerek fonksiyonların aldığı değerler hakkında bilgi edinilir. Tabii bu durum kara kutu fuzzing yaklaşımı benimsenmediğinde geçerlidir.

2. Önceki adımın çıktısı incelenerek kontratın fonksiyonları için geçerli girdiler üretilir. Bu girdiler daha sonra çeşitli değişimlere (mutasyonlara) maruz bırakılarak rastgele hale getirilebilir.
3. Yaratılan tüm değerler ile kontrat çalıştırılır ve çıkan sonuçlara bakılır. Çıkan sonuçlardan zafiyetler ayıklanır ve kullanıcıya sunulur.

Yönteme yaklaşımları gereği üç tip fuzzing yöntemi vardır. Bunlar sırasıyla kara kutu, beyaz kutu, gri kutu fuzzing yöntemleridir.

- **Kara kutu fuzzing:** Kontrat hakkında (içerdiği fonksiyonlar, alabildiği girdiler vs.) herhangi bir bilgi sahibi olmadan tamamen rastgele girdiler kullanılarak yapılan testlerdir. Tamamen rastgele girdiler kullanılarak kontrattaki zafiyetleri keşfetmek amaçlanır. Diğer akranlarına göre daha pratik ve daha hızlı sonuç vermesine karşın daha ilkel bir yöntemdir. Aynı zamanda yeni dallanmaları keşfetme, işlenen kodun genişliği ve sonuç efektifliği açısından da akranlarına göre geridedir [62].
- **Beyaz kutu fuzzing:** Beyaz kutu yaklaşımı kara kutu yaklaşımının tam tersi bir anlayış benimser. Bu da demek oluyor ki bu yaklaşımda kontrat hakkında bilgi edinilip, edinilen bilgiler ışığında girdiler oluşturulur. Bu bilgi edinme işlemi kontratı sürekli olarak sembolik ve gerçek girdiler ile çalıştırarak, kontların kısıtları ve dallanmaları hakkında bilgi edinir. Bu gereksinimler doğrultusunda üretilen girdiler kullanılarak kontrat test edilir [62]. Bu yöntem kara kutu yaklaşımına göre daha geniş bir kapsam sunuyor olsa da oluşan her dallanma için kısıtlamaların ve bu kısıtlamalara uygun girdilerin belirlenmesi test süresinin uzamasına yol açar.
- **Gri kutu fuzzing:** Gri kutu fuzzing yaklaşımı daha önce bahsedilen iki yaklaşımın da belirli özelliklerini taşımaktadır. Bu sayede verdiği sonuçlar da iki yaklaşımın ortalaması niteliğindedir. Yani gri kutu fuzzing yaklaşımında kara kutu yaklaşımının aksine program hakkında bilgi sahibi iken girdiler üretilir. Ancak buradaki kapsam beyaz kutu yaklaşımında anlatıldığı kadar detaylı değildir. Bahsedilen kapsamı tarayabilmek için gri kutu fuzzing araçları hafif enstrümantasyon denilen bir teknikten faydalanır. Gri kutu fuzzing işleminin çalışma şekli hakkında detaylı bilgi şekil 4.1'de bulunabilir. Algoritma 1'in çalışma mantığı şu şekildedir [63]:

1. Fuzzig aracı, test etmek için *prog* adlı programı ve bu program için girdi olarak kullanılacak olan tohum seti *S'*i girdi olarak alır ve test işlemine başlar. Programın her yürümesinde enstrümantasyon o an çalıştırılan dallanmayı yakalayıp eşi olmayan bir ID ile eşleştirebilir.
2. Daha sonra mutasyona uğrayacak bir girdi seçilir.
3. Daha sonrasında ise seçilen girdiye bir *enerji* atanır. Burada enerji değeri seçilen girdinin kaç kere fuzzlanacağını gösterir.
4. Girdi mutasyona uğratılır ve bu değerle program çalıştırılır.
5. Program yeni bir dallanma keşfederse, buna yol açan girdi tohum setine eklenir.
6. Bu işlemler zaman aşımı, maksimum tohum sayısına ulaşılması gibi yürütme sınırlarına ulaşana kadar tekrarlanır

```
Input: Program prog, Seeds S
PIDs ← RunSeeds(S, prog)
while ¬Interrupted() do
        input ← PickInput(PIDs)
        energy ← 0
        maxEnergy ← AssignEnergy(input)
        while energy < maxEnergy do
                input′ ← FuzzInput(input)
                PID′ ← Run(input′, prog)
                if IsNew(PID′, PIDs) then
                        PIDs ← Add(PID′, input′, PIDs)
                energy ← energy + 1
Output: Test suite Inputs(PIDs)
```

Şekil 4.1: Gri Kutu Fuzzing [63].

Akıllı kontratlar için dinamik analiz gelişmekte olan bir alan olsa da bu alanda pek çok araç bulmak mevcuttur. Öne çıkan bazı dinamik analiz araçları şunlardır:

- **Echidna:** Echidna Haskell kullanılarak yazılmış bir fuzzing aracıdır. Şu an piyasada akıllı kontratlar için bulunan en popüler fuzzing araçlarından biridir. Bunun sebeplerinden bazıları sağlamış olduğu güçlü grafik ara yüz ve sürekli entegrasyon (CI) desteği ile özel test yazmaya izin veren yapısı olarak gösterilebilir [64].
- **Maian:** Maian Python da yazılmış otomatik bir akıllı kontrat test aracıdır. Dinamik sembolik yürütme yöntemini baz alan araç, bulduğu zafiyetleri kontratları üç kategoriye ayırır; açgözlü (greedy), müsrif (prodigal), intihara

meyilli (suicidal). Sembolik yürütme ve somut doğrulama olmak üzere iki ana bileşeni vardır. Sembolik yürütmeden çıkan sonuçlar somut doğrulamada doğrulanır [65].

- **ContractFuzzer:** ContractFuzzer girdi olarak aldığı akıllı kontratlardaki ABI'ları kullanarak fuzzing için girdi üreten bir araçtır. Test için kahinler belirler ve EVM enstrümantasyonu ile akıllı kontratların çalıştıkları zamanki davranışlarını kaydeder. Daha sonra bunları kullanarak zafiyetleri tespit eder. 6991 kontrat üzerinde yapılan testlerde 459'un üzerinde zafiyet bulunmuştur [66].

4.3.2.1. M1: Sonuç efektifliği

Dinamik analiz benimsemiş olduğu yaklaşımı bir kez daha hatırlatmak gerekirse, bu teknik kontratların davranışlarını inceler. Bir açıdan bu yaklaşımı gerçek bir saldırganın kontratlara saldırmasına benzetmek mümkündür. Kontratın statik olarak kodunu incelemek yerine, kontrat ile etkileşime girerek onun davranışlarını gözlemleyerek ilerlemesi dinamik analizlerin statik analiz yöntemlerine göre daha uzun sürede tamamlanmasına yol açar. Ancak sürede ortaya çıkan bu kayıp sonuçlardaki düşük hatalı negatif değerleri ile telafi edilir. Bu durum dinamik analiz yöntemlerinin çoğu zaman statik analiz ile elde edilen sonuçların doğrulanması için kullanılmasına yol açmaktadır. Bu anlatılanlar ışığında dinamik analizlerin hatalı negatif üretmediği gibi bir çıkarımda bulunmak son derece yanlış olur. Her teknik gibi dinamik analiz de hatalı pozitif ve negatif değerler üretmektedir. Buna bir örnek Palina Tolmach, Yi Li Shang-Wei Lin, Yang Lui ve Zengxiang Li'nin *A Survey of Smart Contract Formal Specification and Verification* [67] isimli çalışmasında bulunabilir. Bu çalışmaya göre *SODA* isimli bir çalışma zamanı doğrulama aracının ürettiği hatalı pozitif değerleri aracın tespit mekanizmasındaki karar verme modellerinden kaynaklanmaktadır.

Hatırlanacağı gibi iki çeşit çalışma zamanı doğrulama yöntemi mevcuttur. Bunlardan ilki ve sıkça tercih edileni kontratın davranışlarını ağ üzerinde izlemek ve zararlı olduğu düşünülen işlemleri tespit etmek ve reddetmek, daha az tercih edilen olan kontratlara koruyucu kodlar yerleştirmektir. Kontratlara koruyucu kodlar yerleştirilmesi ekstra gaz tüketimine yol açmaktadır [67]. Şu ana kadar bu maddede kontratların sonuç efektiflikleri incelenirken süre ve verdikleri sonuçların kesinliği

üzerinden hareket edilmiş olsa da gaz tüketimi de sonuç efektifliğini etkileyen önemli bir faktördür. Çalışmanın giriş kısmında anlatılan gaz yapısı göz önünde bulundurulduğunda ekstra gaz tüketimi kontrat sahiplerine ekstra para harcatmak anlamına gelir. Söz konusu bu durum genelde analiz yöntemlerinde çok karşılaşılan bir durum olmasa da sonuçları bakımından diğer etmenlere göre daha ciddidir.

4.3.2.2. M2: Teknik kısıtlar

Sembolik yürütmenin teknik kısıtlarının bir kısmından statik analiz başlığı altında bahsedilmişti. Bunlara ek olarak dinamik sembolik yürütme özet fonksiyonlarını işlerken sıkıntı yaşamaktadır [62]. Buradaki durum statik sembolik yürütmede bahsedilen döngüler ile ilgili soruna benzemektedir ve sembolik yürütme motorlarının olaya yaklaşım biçiminden kaynaklanmaktadır.

Fuzzing yöntemlerinin en büyük sorunları program dallanmalarını keşfederken yaşanmaktadır. Bunlar aynı dalın birden fazla kere test edilmesi ve programdaki tüm dallanmaların keşfedilememesi olarak isimlendirilebilir. Bu sorunların çıkış noktası rastgele girdi üretme aşamasıdır. Rastgele üretilen girdiler kontrat üzerinde test yürütürken daha önceden keşfedilmiş bir dalı ortaya çıkarıp bunun tekrar test edilmesine yol açabilir. Fuzzing işlemleri genelde bir zaman aşımı veya maksimum tohum değerine ulaşınca sonlanır. Bu sebeple aynı dalı test etmeyi amaçlayan girdiler kontrattaki farklı ve daha önce keşfedilmemiş dallanmaları keşfetmeyi önleyebilir ki bu da o dallardaki olası zafiyetlerin keşfedilmemesi anlamına gelir. Tabii bu sorunların ciddiyeti ve ortaya çıkması tercih edilen fuzzing yöntemine göre farklılık göstermektedir. Örneğin kara kutu fuzzing yönteminde bu durum beyaz kutu yöntemine göre çok daha fazla gerçekleşme ihtimaline sahiptir. Gri kutu yöntemi her iki fuzzing yönteminin ortasında optimum sonuçlar veren bir yaklaşım olmasına rağmen o da bu durumdan mustariptir [62].

Çalışma zamanı yürütme yönetiminin benimsediği popüler yaklaşım olan kontratı ağ üzerinde izleme yaklaşımı Ethereum gibi blokzincirlerde mecburi çatallanma (hard fork) denilen ve genellikle tercih edilmeyen bir işleme neden olmaktadır [68]. Bunun sebebi performans giderlerinden kaynaklanmaktadır. Daha iyi sonuç veren, daha düşük hatalı negatif oranlarına sahip sonuçlar elde etmek için çalışma zamanında devreye girecek belirli kontroller eklemek gerekir. Bu kontroller de mecburi çatallanma (hard fork) işlemine yol açmaktadır.

4.3.3. Araç ile formül ile doğrulama

Formül ile doğrulama yöntemleri akıllı kontratların analizinde kullanılan en yeni ve piyasada en az yer kaplayan tekniktir. Formül ile doğrulama yöntemi akıllı kontratlardan bağımsız olarak bir tasarımın dizayn aşamasındaki özellikleri taşıyıp taşımadığını kontrol etmek amacıyla yapılan kontrollere denir. Bu kontroller genelde teorem kanıtlayıcılar veya matematiksel yaklaşımlar kullanılarak kontrol edilir [69]. Akıllı kontratlarda formül ile doğrulama işlemi yapılırken genellikle kontratın şu özellikleri kontrol edilir; fonksiyonel tutarlılık, çalışma zamanı güvenliği, güvenilirlik ve sağlamlık. İki tane popüler formül ile doğrulama yöntemi diğerlerinin arasından öne çıkmaktadır. Bunlar *teorem ile doğrulama* ve *program doğrulama* yöntemleridir.

Teorem ile doğrulama yöntemi kontratı ve kontratın tahmini değerlerini bir matematiksel formül olarak kodlar. Daha sonra bu kodlamadan, kontratın aksiyomlarına ve çıkarım kurallarına bağlı kalınarak bir ispat türetilir [67].

Program doğrulaması ise genellikle akıllı kontratların kaynak kodlarının doğrulama araçlarının kullandığı programlama dillerine çevrilmesi ve daha sonra bu araçlarda bulunan ispat motorları kullanılarak çevrilen kodun incelenmesi şeklinde çalışmaktadır.

Formül ile doğrulama tekniğinde öne çıkan araçlar şu şekildedir:

- **F*:** Esas olarak fonksiyonel doğruluk ve çalışma zamanı güvenliğini test etmeyi amaçlayan bu araç, kontratların kaynak veya bayt kodlarını fonksiyonel bir dil olan F*'a çevirir. F* aracının ispat motorunun içinde *Solidity** ve *EVM** adlı iki yapı bulunur. Bu yapılar çevrilmiş olan kod üzerinde analiz işlemlerini gerçekleştirir [2].
- **Isabelle/HOL:** Kontratın tüm elementlerin ayrı bir durum tarafından ifade edildiği bu araçta, akıllı kontratın davranışları ile ilgili güvenilir sonuçlar elde etmek mümkün. Doğrulama işlemi kontratın bayt kodu üzerinde yapılır. Araç aldığı kodu ön yükleyici ve çalışma zamanı kodu olarak ikiye ayırır. Ön yükleyici kodu kontratı bloğa yüklemek, çalışma zamanı kodu ise kontratın fonksiyonlarını test etmek için kullanılır. Bu yaklaşım sayesinde araç kısa bir kontratta bile detaylı sonuçlar verebilmektedir [2].

4.3.3.1. M1: Sonuç efektifliği

Akıllı kontratlar için yeni gelişmekte olan bir alan olmasına karşın formülü ile doğrulama tekniği yazılım dünyası için yeni bir teknik değildir ve bu alanda pek çok araç ve yaklaşım mevcuttur. Yukarıda anlatılan yöntemler ve verilen örnekler incelendiğinde bunların bir kısmının (HOL gibi) özünde akıllı kontratlar için ortaya çıkmamış olduğu görülebilir. Ancak bu yaklaşımlara yapılan birkaç ekleme ve değişiklik ile bunların akıllı kontratlar üzerinde çalışması sağlanmıştır. Bu değişikliklere örnek olarak kontratların kodlarının araçların çalışma dillerine çevrilmesi ve ispat motorlarının uygun şekilde değiştirilmesi verilebilir. Tabii araçlar bunu yaparken kontratın el değmemiş halindeki sağlamlığı korumalıdır aksi takdirde çıkan sonuçların doğruluğu tartışmaya açık olur. Bu sağlamlık durumunu Schneidewind ve ark. yaptıkları çalışmada incelemiş ve çıkan sonuçların iddia edildiği gibi sağlamlık sunmadığı iddia etmişlerdir [67].

Formül ile doğrulamanın sonuçlarını etkileyen bir başka durum da girdi olarak verilen kontratın yazılmış olduğu dildir. Formül ile doğrulama girdi olarak üç çeşit programlama dilinden birini kabul etmektedir. Bunlar yüksek-seviye diller (Solidity, Go vs.), düşük seviye diller (Bitcoin Script) ve orta-seviye diller (bayt kodu) olarak sınıflandırılabilir. Farklı girdiler üzerinde yapılan çalışmalarda orta seviye dillerin verdikleri sonuçlar bakımından formül ile doğrulama yaklaşımına en uygun diller oldukları görülmüştür [70].

4.3.3.2. M2: Teknik kısıtlar

Formül ile doğrulama yöntemlerine genel olarak bakıldığında teknik kısıtlardan ilk göze çarpanı teorem ile doğrulamanın tam olarak otomatik olmayışıdır. Bu yöntemin otomatik olmamasının nedeni insan uzmanlığına ihtiyaç duyulan yerlerin fazlalığıdır.

Bir önceki maddede de bahsedildiği gibi formül ile doğrulamada özellikle program doğrulaması yönteminde kontratın başka bir dile soyutlanması sırasında sağlamlık kaygısı ortaya çıkar. Bu sağlamlık kaygısı kontların bellek ve çalışma bileşenlerinin soyutlama işlemi sırasında doğru bir şekilde aktarılamamasından kaynaklanır. Bu durumun sonuçlarında üstteki maddede kısa bir şekilde bahsedilmiştir.

4.3.4. Sonuç ve Tartışma

Akıllı kontratlarda zafiyet analizi yapmak için tercih edilen üç popüler teknik vardır. Bunlar statik analiz, dinamik analiz ve formül ile doğrulamadır. Yukarıda bu teknikler ve bunların başlıca kullandıkları yöntemler hakkında incelemeler mevcuttur. Bu incelemeler tekniğin sunduğu sonuç efektifliği ve teknolojinin teknik kısıtları göz önünde bulundurularak yapılmıştır.

İlk incelenen teknik olan statik analiz kabaca bir akıllı kontratın kaynak veya bayt kodu üzerinde kontratı çalıştırmadan yapılan analiz işlemleri olarak tanımlanabilir. Bu tür analizler kod üzerinde yapılan hata ayıklama işlemlerine benzetilebilir. Bu analiz yöntemi akıllı kontrat camiasında çokça tercih edilen ve oturmuş bir yapıya sahiptir. Ayrıca şu ana kadar tanımlanmış olan zafiyetleri keşfetme konusunda çok etkilidir. Ancak benimsemiş olduğu işleyiş biçimi sebebiyle daha önce karşılaşılmamış zafiyetleri keşfetmesi mümkün değildir. Buna ek olarak başlıkta bahsedilen diğer tekniklere göre daha yüksek hatalı pozitif ve negatif değerlerine sahipler sonuçlar sunmaktadır. Hatırlanacağı gibi bir önceki bölümde denetimleri yapılmasına karşın hacklenmiş kontratlardan bahsedilmiş ve bu çalışmanın amacının bu tarz olayları minimize etmek olduğundan bahsedilmiştir.

Dinamik analiz yöntemi ise statik analizin aksine kontratları çalıştırarak veya çalıştıkları ortamda incelemeyi tercih eder. Bir nevi kontrata saldıran gerçek bir saldırgan gibi çalışırlar ve bu sayede kontratın etkileşim halindeki davranışlarını da gözlemleme şansı sunar. Benimsenen bu yaklaşım statik analiz yöntemine göre daha uzun sürede sonuç veriyor olsa da daha düşük hatalı pozitif ve hatalı negatif değerleri sunarak daha tutarlı ve kesin sonuçlar vermektedir. Dinamik analiz yöntemleri kendi aralarında karşılaştırıldığında fuzzing yöntemi öne çıkmaktadır. Üç farklı çeşidi olan bu yöntem dinamik analiz tekniğinin diğer yöntemleri ile benzerlikler göstermektedir. Buna örnek olarak dinamik sembolik yürütme ve beyaz kutu fuzzing verilebilir. Ancak kontratın dallanmalarının keşfi sırasında aynı dalın birden fazla kere test edilmesi fuzzingnın karşılaştığı ciddi bir problemdir. Ancak tüm bunlara karşın fuzzing, özellikle gri kutu fuzzing, zafiyet tespiti için önemli bir adaydır.

Formül ile doğrulama akıllı kontratlar için gelişmekte olan bir teknik olarak kabul edilebilir. Formül ile doğrulama genel olarak gerekirse bir programın istenilen özellikleri (genelde bakılan özellikler fonksiyonel tutarlılık, çalışma zamanı

güvenliği, güvenilirlik ve sağlamlık) taşıyıp taşımadığını anlamak için yapılan kontrollerdir. Bu yöntemdeki araçlar aslında direkt olarak akıllı kontratlar için üretilmemiş belirli modifikasyonlar sonucu akıllı kontratlar için çalışabilecek seviyeye getirilmişlerdir. Aynı zamanda tekniğin belirli yöntemlerinin tam olarak otomatik edilememesi ve insan denetimine ihtiyaç duyması da formül ile doğrulamanın zafiyet tespitinde kullanılmasını güçleştirmektedir.

Toparlamak gerekirse araç analizini tek başına zafiyet tespiti için kullanmak yetersiz olacaktır. Özellikle işletme mantığındaki zafiyetlerin tespiti konusunda sadece araçlara güvenmek yetersiz bir denetim sürecine yol açacaktır. Araç analizlerini manuel analiz ile beraber, birbirlerini tamamlayacak şekilde kullanmak mantıklı olacaktır. Bu sayede manuel analizde gözden kaçan noktaları tespit etmek, uç nokta testlerini daha efektif ve detaylı bir biçimde yapabilme yetisi edinilebilir. Statik analiz yöntemleri (statik sembolik yürütme ve geleneksel statik analiz) kullanılarak gözden kaçan noktalar tespit edilebilir, aynı zamanda manuel analizdeki hatalı pozitif değerleri elimine edilebilir. Bu başlıkta daha önce bahsedildiği gibi dinamik analiz sonuçları statik analiz sonuçlarını doğrulamak için kullanılabilir aynı zamanda uç nokta testleri için de dinamik analiz yöntemlerinden faydalanmak mantıklıdır. Burada anlatılanlara göre dinamik sembolik yürütme ve gri kutu fuzzing yöntemleri bahsedilen amaçlarla dinamik analiz kullanımına en uygun olanları olarak düşünülmüştür. Ancak burada unutulmaması gereken nokta seçilen dinamik ve statik analiz yöntemleri ve araçları denetim işlemini yürüten kişinin tercihine ve alışkın olma durumuna göre değişiklik gösterebilir. Ancak iki yöntemin manuel analizle bir arada kullanılması değişkenlik göstermemelidir.

4.4. Raporlama ve Sonrası

Önerilen metodolojinin son aşaması süreçte bulunan bulguların doğru ve detaylı bir biçimde raporlanmasını kapsamaktadır. Yürütülen denetim süreci her ne kadar detaylı ve iyi olursa olsun raporlaması iyi yapılmadığı takdirde, bu süreç karşı tarafa aktarılamayacağından yapılan denetimin etkisi azalır. Öyle ki raporlama iyi değilse belirli bulgular raporda yer alsa bile tam olarak düzeltilemez. Kısacası raporlama bir nevi tüm süreç boyunca yaşananları proje yetkililerine anlatmak gibi düşünülebilir.

Önerilen raporlama şekli ile devam etmeden önce denetim işlemleri ile ilgili akıldan çıkarılmaması gereken iki önemli noktaya değinmek yerinde olacaktır.

Bunlardan ilki denetim işleminin amacı projenin kodunu değerlendirerek proje yayınlanmadan önce potansiyel saldırı yüzeylerini tespit ederek, bu yüzeylerin kapatılmasını sağlamaktır. Diğer önemli nokta ise denetim işlemlerinin projeyi kullanacak kullanıcılar için değil proje ekibi için yapılan işlemlerdir.

Bu noktalara gözetilerek hazırlanan raporda şu maddeleri içermelidir:

- Kapsam: Raporun bu maddesinde proje sahiplerinin denetlenmesini istediği kısımlar yer alır.
- Zaman Çizelgesi: Her proje için farklıdır. Denetim sürecinin zamansal yayılımını gösterir.
- Amaç: Yukarıda anlatılanlar doğrultusunda denetim işleminin amacının tanımlandığı maddedir.
- Projenin Genel Bilgileri: Bu madde projedeki rolleri, yetkileri, varlıkları içerir.
- Kullanılan araçlar ve yöntemler: Denetim sürecinde kullanılan otomatik araçları, test ortamı için kullanılan teknolojileri ve diğer analiz yöntemlerini içerir.
- Bulguların genel özeti: Bu madde denetim işlemi sırasında tespit edilen bulguların genel özetini içerir. Bulgular sınıflandırılarak listelenmelidir. Bu sınıflandırma işlemi kişi ve kurumlara göre değişkenlik gösterse de genellikle yüksek, orta, düşük, bilgilendirme şeklinde ayrılır. Bu metodoloji de bu yaklaşımı benimsemiştir. Bulguların sınıflara ayrılması için önem puanı yaklaşımı yaygındır. Önem puanı hesaplamak için pek çok yaygın metot bulunmaktadır, bunlardan en yaygın olanı OWASP Risk Derecelendirme Metodolojisidir[1]. Bu metodolojide OWASP yaklaşımına benzer bir biçimde bulgunun ciddiyet derecesi ve yaşanma şansı 5 üzerinden puanlanıp, toplanacaktır çıkan sonuç aşağıdaki gibi sınıflandırılacaktır:
    - 1-3: Bilgilendirme
    - 3-6: Düşük

[1] https://owasp.org/www-community/OWASP_Risk_Rating_Methodology

- 6-8: Orta
- 8-10: Yüksek

Burada dikkat edilmesi gereken nokta bulgunun ciddiyet derecesi ve yaşanma olasılığının puanlanmasıdır. Bu metodoloji yaşanma şansı puanlanırken aşağıdaki gibi davranır:

- 3-5: Eğer zafiyet erişim fazla kaynak gerektirmeden birkaç işlemde tetiklenebiliyorsa,
- 2-3: Eğer zafiyet erişim izni, uygun rol, sistemin çalışması hakkında detaylı bilgi ve kaynak gerektiriyorsa,
- 1-2: Yukarıda anlatılanlardan daha zor şartlar ile tetiklenebilen bir zafiyetleri içerir.

Zafiyetin ciddiyeti aşağıdaki gibi puanlanır:

- 3-5: Projedeki Ehter'in kaybına veya erişilmez hale gelmesine, projenin sahipliğinin değişmesine yol açan zafiyetler,
- 2-3: Projenin normal akışının dışında davranışlar göstermesine yol açan ancak Ether kaybına yol açmayan zafiyetler,
- 1-2: Yukarıda anlatılanların dışında kalan türdeki zafiyetleri kapsar.

- Bulguların detayları: Bu maddede bulgular hakkında detaylı bilgiler verilir. Genel olarak bulguların hangi durumlarda ve neden ortaya çıktıları, nelere yol açabilecekleri, projenin hangi kısımlarında (kaynak kod satırı) bulundukları, bulguları gidermek için neler yapılabileceği, bir önceki maddede anlatılan ciddiyet ve yaşanma şansı puanları detaylı bir şekilde açıklanması gereken noktalardır. Üçüncü bölümde çeşitli zafiyetler örnekler ve bu zafiyetlerin nasıl giderilebileceği anlatılmıştır. Bu bölüm için üçüncü bölümdeki yapıya benzer bir yapı düşünülebilir.
- Sömürü Senaryoları: Eğer sömürü senaryosu uygulanmış ise bu bölümde ekran görüntüleri ile senaryo adım adım anlatılmalıdır. Kurulan test ortamından bahsederek bu aşamaya başlamak da unutulmaması gereken bir noktadır. Aynı zamanda sömürü senaryosunun kanıt videosu da eklenebilir.

Bu noktalara dikkat ederek hazırlanmış örnek bir rapor taslağını ekte bulmak mümkündür.

Rapor teslimi yapıldıktan sonra proje yetkilileri gönderilen raporu inceleyerek, bazı bulguları (veya hepsini) düzeltip denetim işlemini gerçekleştiren tarafa yapılan düzeltmelerin kontrol edilmesi talebi ile gelebilirler. Burada yapılması gereken proje yetkililerinin hangi bulguları düzelttiğini öğrenip bu çalışmada anlatılan analiz yöntemlerinden seçilen yöntemler ile bu düzeltmeleri doğrulamaktadır. Bu aşama zorunlu bir aşama olmamakla beraber, yaygındır. Ancak zaman olarak çok kısa süreceğinden çoğu zaman denetim işlemini yürüten taraf için bir problem teşkil etmez.

4.5. Sonuç

Bu bölümde anlatılanları özetlemek gerekirse, önerilen metodolojinin dört aşaması vardır; bilgi edinme, manuel analiz, araç analizi, raporlama ve sonrası. Bilgi edinme aşamasında denetim sürecinin başında proje hakkında deneticinin bilgi edinmesini sağlar. Bu bilgiler projenin ne yapmayı amaçladığı, hangi yapı ve teknolojileri kullandığı, işletme mantığı, projenin nasıl implemente edildiği olarak sınıflandırılabilir. Manuel analiz ise iki basamaklı bir yapıdır; manuel inceleme ve manuel test. Manuel inceleme basamağında denetçi proje kaynak kodları üzerinden geçer ve olası zafiyetleri tespit etmeye çalışır. Burada denetçinin hangi noktalara dikkat etmesi gerektiği detaylı bir biçimde anlatılmıştır. Manuel test basamağında ise projenin hali hazırda yapılmış olan testleri incelenir. Buna ek olarak proje bir test ortamına yüklenir ve manuel incelemede şüphelenilen noktalar ve uç noktalar için test senaryoları yazılır. Ayrıca bu basamakta çeşitli bulguların daha iyi anlaşılması için kanıt sömürü senaryoları uygulanabilir. Araç analizi aşamasında ise çeşitli otomatik araçlar kullanılarak manuel analizde olduğu gibi statik ve dinamik testler uygulanır. Araç analizinin üç farklı çeşidi vardır: statik, dinamik ve formül ile doğrulama. Bu çeşitler teknik kısıtlar ve sonuç efektifliği bakımından incelendiğinde. Statik ve dinamik araç analizi yöntemlerinin kullanılmasın makul olduğu ortaya çıkmıştır (Daha spesifik olmak gerekirse geleneksel statik analiz, sembolik yürütme ve gri kutu fuzzing yöntemleri). Ancak araç analizinin tek başına bilgi edinmeden sonra tek başına kullanılmasının yetersiz olduğu sonucu da ortaya çıkmıştır. Bunun sebebi araç analizinin tek başına işletme mantığındaki ve geliştiricilerin gözden

kaçırmasından kaynaklanan bulguları tespit etmedeki yetersizliğidir. Raporlama ve sonrası aşamasında ise tespit edilen bulguların ve sürecin proje yetkililerine doğru bir şekilde aktarılması için neler yapılması gerektiği anlatılmıştır. Buradan da anlaşılabileceği gibi bilgi edinme ve raporlama aşamaları bulgu tespitini destekleme amacındadırlar.

Akıllı kontratlarda efektif bir denetim yapabilmek için en önemli nokta manuel analiz işleminin doğru yapılmasıdır. Manuel analiz yöntemi her ne kadar zaman açısından araçla incelemeye göre daha pahalı olsa da sunduğu sonuçlar göz önünde bulundurulduğunda buna değer. Çünkü manuel analiz işlemi alt yapı bazlı zafiyetlerin yanı sıra işletme mantığındaki zafiyetlerin ve geliştiricilerin es geçtiği noktaların tespitine de olanak tanır. Bu noktaları otomatik araçlar ile tespit etmek hem araçların yapıları gereği uygun değildir hem de çok fazla hatalı pozitif ve negatif değer üretir. Yalnız bu kesinlikle otomatik araçların kullanılmasının gereksiz olduğu gibi algılanmamalıdır. Otomatik araçlar, manuel analiz yöntemlerini destekleyici bir biçimde sürece dahil edilmelidir. Bunun iki sebebi vardır. Birincisi her ne kadar manuel süreçler kapsamlı olsalar da kapsamları süreçleri yürüten kişinin birikimine dayandığından mutlaka gözden kaçan noktalar veya gereksiz bulgular olacaktır. İkincisi de projenin uç nokta testleri yapılırken manuel süreçlerden ziyade otomatik süreçler hem daha hızlı sonuç verir hem de aynı sürede daha geniş bir kapsamlı bir denetim yapma imkânı taşır.

5. SONUÇ VE ÖNERİLER

Güncel blokzincir teknolojisinde akıllı kontratlar sıklıkla kullanılmaktadır. Bunun nedeni yapıları gereği sağladıkları otomasyon ve güvenliktir. Ancak her teknolojide olduğu gibi akıllı kontratlarda da güvenlik açıkları bulunmaktadır. Bu zafiyetler gerek kontrat yaratıcılarının gözden kaçırdığı noktalardan ve eksik varsayımlarından, gerek kullanılan yapıtaşlarının yetersizliklerinden kaynaklanmaktadır. Her ne kadar bu zafiyetler için çözümler üretilmeye çalışılsa da diğer tüm teknolojilerde olduğu gibi yeni zafiyetler ortaya çıkmaktadır. Bu sebeple akıllı kontratlardan faydalanan projelerin güvenliğini sağlamak için denetim denilen prosedürler uygulanmaktadır. Bu prosedürler, projeyi değerlendirerek potansiyel saldırı yüzeylerini tespit edip, tespit edilen yüzeylerin giderilmesini amaçlar. Akıllı kontratlar blokzincir yapısında çalıştıkları için denetim işlemleri projeler blokzincire yüklenmeden önce yapılmalıdır. Denetim servisi sağlayan pek çok kurum ve kişi bulmak mümkündür ve hepsinin kendilerine göre takip ettikleri bir akışları mevcuttur. Denetim işlemleri her ne kadar saldırı yüzeylerini tespit edip engellemeyi amaçlasa da kimi zaman denetim (veya denetimler) uygulandıktan sonra bile projeler atakların hedefi olup ciddi kayıplar yaşamışlardır.

Az önce bahsedilenler doğrultusunda, bu çalışmada bahsi geçen kayıplara neden olan olaylardan Ethereum ekosistemi için önemli olanları, ekosistemde sıklıkla rastlanan zafiyetler ile beraber incelenmiştir. Bu incelemenin amacı kapsamlı bir denetim metodolojisi önererek kayıpların önüne geçmeye çalışmaktır. Ekosistemde ses getirmiş ve önceden denetim uygulanmış olaylar incelendiğinde, bu olayların büyük çoğunluğunun önüne geçebilmenin mümkün olduğu görülmüştür. Bunun nedeninin ise söz konusu işletme mantığı incelenirken gözden kaçan noktalar ve sadece otomatik araçlara güvenen süreçler olduğu gözlemlenmiştir. Buna ek olarak çalışmada incelenen zafiyetlerde etkin zafiyetlerin %60'ının kullanıcı ihmallerinden kaynaklandığı görülmektedir. Bu tarz zafiyetlerin tespiti mantıksal eksikliklere göre daha kolaydır, hem araçlar hem insanlar için. İncelenen zafiyetler hakkında bahsedilmesi gereken nokta, Ethereum ekosisteminde bilinen ve süreklilikle çıkan reentrancy, zayıf rastgelelik (weak randomness) gibi zafiyetlere ek olarak çalışmanın güncelliğini korumak için 2020 yılında en çok karşılaşılan zafiyetler de incelemeye dahil edilmiştir. Bu ilginç bir sonuca yol açmıştır. Reentrancy, zayıf rastgelelik, blok

değişkenlerinin kullanımı, kontrol edilmeyen alt seviye çağrılar gibi bilinen zafiyetler 2020 yılında da en sık rastlanan zafiyetler arasında yer almışlardır (Tabii bunlara ek olarak yeni olarak nitelendirilebilecek zafiyetler görmek de mümkündür).

Toparlamak gerekirse manuel inceleme ile araçları bir arada kullanmayı amaçlayan önerilen metodoloji ile yukarıda bahsedilen zafiyet ve olayların önlenebileceği düşünülmektedir. Çünkü önerilen metodoloji manuel analizi baz alır. Araç analizini ise manuel analize yardımcı olacak şekilde kullanır. Bu sayede manuel analiz ile dil bazlı ve altyapısal zafiyetlerin yanı sıra mantık hatalarını ve eksik noktaları da tespit etmek mümkün olacaktır. Ancak unutulmamalıdır ki manuel analiz insan bazlı bir süreçtir, sonuçlarında mutlaka atlanan noktalar veya hatalı bulgular olacaktır. Bu sebeple hem bu eksiklikleri önlemek hem de uç nokta testleri gibi zaman alan geniş kapsamlı testleri düzgün şekilde yapabilmek için araç analizi de manuel analize ek olarak kullanılmalıdır. Başka bir deyişle, anahtar nokta bu iki yaklaşımın birbirlerini eksiklerini kapatacak şekilde doğru bir biçimde kullanılmasını sağlamak olacaktır. Bu tezde bu şekilde çalışan bir metodoloji önerilmiştir. Bu metodoloji aynı zamanda manuel analiz sırasında bir denetimcinin dikkat etmesi gereken noktaları (2020 zafiyetleri ve manuel analiz başlığındaki maddeler) kontrol listesine benzer bir yapı gibi sunmayı amaçlamıştır.

5.1. Gelecek Çalışmalar

Ethereum ekosisteminin güvenliği gelişmekte olan dinamik bir alandır. Bu sebeple bu çalışma pek çok şekilde geliştirilebilir.

- Önerilen metodolojinin kapsamını genişletmek için Ethereum ekosisteminde geriye dönük ve ileri dönük tehdit modellemeleri yapılabilir.
- Önerilen metodoloji bir iki senelik bir süreçte denenerek metodolojideki eksiklikler ve fazlalıklar tespit edilip metodolojinin evrilmesi makuldür. Burada fazlalıklardan kasıt efektif olmayan akışların ve araçların daha efektif olanları ile değiştirilmesidir. Eksikliklerden kasıt ise gelişen Ethereum teknolojisinde ortaya çıkan akıllı krediler gibi yeni özelliklere, yeni önemli olay ve zafiyetlere metodolojinin uyum sağlamasındır.

KAYNAKLAR

[1] **Atzei N., Bartoletti M., and Cimoli T**., "A Survey of Attacks on Ethereum Smart Contracts (SoK)," Lecture Notes in Computer Science Principles of Security and Trust, pp. 164–186, 2017.

[2] **Praitheeshan P., Pan L., Yu J., Liu J.,and Doss R.,**"SecurityAnalysis Methods on Ethereum Smart Contract Vulnerabilities: A survey," *arXiv.org*, 16-Sep-2020. [Online]. Available: https://arxiv.org/abs/1908.08605.

[3] **Samreen N. Fatima and Alalfi M. H.**, "Reentrancy Vulnerability Identification in Ethereum Smart Contracts," 2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE), 2020, pp. 22-29, doi: 10.1109/IWBOSE50093.2020.9050260.

[4] **Sayeed S., Marco-Gisbert H. and Caira T.,** "Smart Contract: Attacks and Protections," in IEEE Access, vol. 8, pp. 24416-24427, 2020, doi: 10.1109/ACCESS.2020.2970495.

[5]"**Smartcontractvulnerabilities:VulnerabledoesnotImplyExploited**." [Online]. Available: https://www.usenix.org/system/files/sec21summer_perez.pdf.

[6] **Cao Y., Zou C., and Cheng X.,** "Flashot: A Snapshot of Flash Loan Attack on defi ecosystem," *arXiv.org*, 01-Feb-2021. [Online]. Available: https://arxiv.org/abs/2102.00626.

[7] **Qin K., Zhou L., Livshits B., and Gervais A.,** "Attacking the DEFI ecosystem with flash loans for fun and profit," *arXiv.org*, 20-Mar-2021. [Online]. Available: https://arxiv.org/abs/2003.03810.

[8] **Bhardwaj, A., Shah, S.B.H., Shankar, A. *et al*.** Penetration testing framework for smart contract Blockchain. *Peer-to-Peer Netw. Appl.* **14,** 2635–2650 (2021). https://doi.org/10.1007/s12083-020-00991-6

[9] **Gaur N., Desrosiers, L., Ramakrishna, V., Novotný, P., Baset, S. and O'Dowd, A.** (2018). *Hands-on blockchain with Hyperledger*

[10] **"The Beacon Chain," *ethereum.org*. [Online]. Available: https://ethereum.org/en/eth2/beacon-chain/. [Accessed: 2022].**

[11] **"Ethereum Whitepaper** | ethereum.org", *ethereum.org*, 2021. [Online]. Available: https://ethereum.org/en/whitepaper/#ethereum. [Accessed: 25- Dec-2020].

[12] **"Ethereum: A secure decentralised ... - github pages." [Online].** Available: https://ethereum.github.io/yellowpaper/paper.pdf.

[13] **Chen J., Xia X., Lo D., Grundy J. and Yang X.,** "Maintaining Smart Contracts on Ethereum: Issues, Techniques, and Future Challenges", *arXiv.org*, 2021. [Online]. Available: https://arxiv.org/abs/2007.00286. [Accessed: 06- Jan- 2021].

[14] **Wang D., Wu S., Lin Z., Wu L., X. Yuan, Zhou Y., Wang H., and Ren K.,** "Towards a first step to understand flash loan and its applications in Defi Ecosystem," *arXiv.org*, 21-Apr-2021. [Online]. Available: https://arxiv.org/abs/2010.12252.

[15] **"Stablemagnet - REKT,"** *rekt*. [Online]. Available: https://rekt.news/stablemagnet-rekt/. [Accessed: 17-Jan-2022].

[16] ***Techrate,*** "Free smart contract audits & development," . [Online]. Available: https://techrate.org/.

[17] **"Harvest finance,"** ***Harvest Finance.*** [Online]. Available: https://harvest.finance/.

[18] **"Harvest Flashloan Economic Attack Post-Mortem,"** ***Medium***, 26-Oct-2020. [Online]. Available: https://medium.com/harvest-finance/harvest-flashloan-economic-attack-post-mortem-3cf900d65217.

[19] ***Grim.Finance*** ,"About Us," . [Online]. Available: https://docs.grim.finance/.

[20] **Furucumbo**, "What is Furucombo?," *Docs*. [Online]. Available: https://docs.furucombo.app/.

[21] "**Furucombo,"** ***rekt.*** [Online]. Available: https://rekt.news/furucombo-rekt/.

[22] **Medvedev E.,** "EthereuminBigQuery:ApublicdatasetforSmartContractAnalytics Google Cloud Blog," *Google*. [Online]. Available: https://cloud.google.com/blog/products/data-analytics/ethereum-bigquery-public-dataset-smart-contract-analytics.

[23] **Crytic**, "Crytic/slither: Static analyzer for solidity," *GitHub*. [Online]. Available: https://github.com/crytic/slither

[24] **Chinen Y., Yanai N., Cruz J. P., and Okamura S.,** "RA: Hunting for Re-Entrancy Attacks in Ethereum Smart Contracts via Static Analysis," *2020 IEEE International Conference on Blockchain (Blockchain)*, 2020.

[4]: **Atzei N., Bartoletti M., and Cimoli T,** 2020 "A Survey of Attacks on Ethereum Smart Contracts (SoK),"

[25] Güçlütürk O. G., "The DAO Hack Explained: Unfortunate Take-off of Smart Contracts," *Medium*, 01-Aug-2018. [Online]. Available: https://ogucluturk.medium.com/the-dao-hack-explained-unfortunate-take-off-of-smart-contracts-2bd8c8db3562.

[26] **Roan A.**, "How Spankchain Got Hacked," Medium, 26-Mar-2020. [Online]. Available: https://medium.com/swlh/how-spankchain-got-hacked-af65b933393c.

[27] **"Security considerations,"** ***Security Considerations -*** *Solidity 0.8.8 documentation*. [Online]. Available: https://docs.soliditylang.org/en/latest/security- considerations.html#use-the-checks-effects-interactions-pattern.

[28] **OpenZeppelin**,"Openzeppelin-contracts/reentrancyguard.solatmaster OpenZeppelin/openzeppelin-contracts," *GitHub*, 30-Aug-2021. [Online]. Available: https://github.com/OpenZeppelin/openzeppelin-contracts/blob/master/contracts/security/ReentrancyGuard.sol

[29] **"SWC-116 · Overview,"** · Overview. [Online]. Available: https://swcregistry.io/docs/SWC-116.

[30] **GovernMental**. [Online]. Available: http://governmental.github.io/GovernMental/.

[31] ***Trail of Bits Blog*****,** "Avoiding smart contract 'gridlock' with Slither," 04-Jul-2019. [Online]. Available: https://blog.trailofbits.com/2019/07/03/avoiding-smart-contract- gridlock-with-slither/.

[32] ***Edgeware****: Old Lockdrop*. [Online]. Available: https://etherscan.io/address/0x1b75b90e60070d37cfa9d87affd124bb345bf70a#code.

[33] **"SWC-132 · overview,"** · *Overview*. [Online]. Available: https://swcregistry.io/docs/SWC-132

[34] **Crytic**, "Detector documentation · Crytic/Slither Wiki," *GitHub*. [Online]. Available: https://github.com/crytic/slither/wiki/Detector- Documentation.

[35] **Khanand Z.A., Namin A.S.**," A survey on vulnerabilities of Ethereum Smart Contracts," *arXiv.org*, 28-Dec-2020. [Online]. Available: https://arxiv.org/abs/2012.14481.

[36] **Altabba M.,** "Hundreds of millions of dollars locked at Ethereum 0x0 address and Smart Contracts' addresses -...," *Medium*, 16-Jun-2018. [Online]. Available: https://medium.com/@maltabba/hundreds-of- millions-of-dollars-locked-at-ethereum-0x0-address-and-smart- contracts-addresses-how-4144dbe3458a.

[37] **"Chapter 9: Smart Contract Security,"** 11-Feb-2021. [Online]. Available: https://www.bookstack.cn/read/ethereumbook-en/spilt.11.c2a6b48ca6e1e33c.md.

[38] ***KotET*** , "Post-mortem investigation (Feb 2016)," *KotET - Post-Mortem Investigation*. [Online]. Available: https://www.kingoftheether.com/postmortem.html.

[39] **Etherpot**, "Contract/lotto.sol at master · etherpot/contract," *GitHub*. [Online]. Available: https://github.com/etherpot/contract/blob/master/app/contracts/lotto.s ol.

[40] **"SWC-104 · overview,"·** *Overview*. [Online]. Available: https://swcregistry.io/docs/SWC-104.

[41] **Reutov** A., "Predicting random numbers in Ethereum Smart contracts," *Medium*, 30-May-2018. [Online]. Available: https://blog.positive.com/predicting-random-numbers-in-ethereum- smart-contracts-e5358c6b8620.

[42] ***reddit*** ,"R/ethereum - smartbillions lottery contract just got hacked!,". [Online]. Available: https://www.reddit.com/r/ethereum/comments/74d3dc/smartbillions_l ottery_contract_just_got_hacked/.

[43] ***OpenAddressLottery***. [Online]. Available: https://etherscan.io/address/0x741f1923974464efd0aa70e77800ba5d9 ed18902#code.

[44] **Sanjuas J.,** "An analysis of a couple Ethereum Honeypot contracts," *Medium*, 18-Feb-2019. [Online]. Available: https://medium.com/coinmonks/an-analysis-of-a-couple-ethereum- honeypot-contracts-5c07c95b0a8d.

[45] **Sigp**, "SIGP/solidity-security-blog: Comprehensive list of known attack vectors and common anti-patterns," *GitHub*. [Online]. Available: https://github.com/sigp/solidity-security-blog#storage.

[46] **Raineorshine**, "Raineorshine/solgraph: Visualize solidity control flow for smart contract security analysis. ⇆," *GitHub*. [Online]. Available: https://github.com/raineorshine/solgraph.

[47] **OpenZeppelin**, "Flash-mintable asset-backed tokens," *OpenZeppelin blog*, 31-Aug-2020. [Online]. Available: https://blog.openzeppelin.com/flash-mintable-asset-backed-tokens/.

[48] ***OpenZeppelin***, "**ERC-20** token standard," *ethereum.org*. [Online]. Available: https://ethereum.org/en/developers/docs/standards/tokens/erc-20/.

[49] ***OpenZeppelin*** ,"ERC721," *OpenZeppelin Docs*. [Online]. Available: https://docs.openzeppelin.com/contracts/2.x/erc721.

[50] **CoinMarketCap**, "ERC-884: CoinMarketCap," *CoinMarketCap Alexandria*, 05-Nov-2021. [Online]. Available: https://coinmarketcap.com/alexandria/glossary/erc-884.

[51] ***OpenZeppelin,*** "ERC621," *EthHub*. [Online]. Available: https://docs.ethhub.io/built-on-ethereum/erc-token-standards/erc621/.

[52] ***OpenZeppelin*** , "ERC721," *EthHub*. [Online]. Available: https://docs.ethhub.io/built-on-ethereum/erc-token-standards/erc721/. [Accessed: 17-Jan-2022].

[53] "**Building-secure-contracts/token_integration.MD** at master · crytic/building-secure-contracts," *GitHub*. [Online]. Available: https://github.com/crytic/building-secure-contracts/blob/master/development-guidelines/token_integration.md#erc-conformity.

[54] ***OpenZeppelin***, ERC20 API: An attack vector on approve/transferfrom methods," *Google Docs*. [Online]. Available: https://docs.google.com/document/d/1YLPtQxZu1UAvO9cZ1O2RPXBbT0mooh4DYKjA_jp-RLM/edit.

[55] **262588213843476, "Token checklist table," *Gist*. [Online].** Available: https://gist.github.com/shayanb/cd495e23c7cf1a8b269f8ce7fd198538#file-token_checklist-md.

[56] **Tian** Z., "Smart Contract Defect Detection Based on Parallel Symbolic Execution," 2019 3rd International Conference on Circuits, System and Simulation (ICCSS), 2019, pp. 127-132, doi: 10.1109/CIRSYSSIM.2019.8935603.

[57] **Smartdec**, "Smartdec/smartcheck: Smartcheck – A static analysis tool that detects vulnerabilities and bugs in solidity programs (ethereum-based smart contracts).," *GitHub*. [Online]. Available: https://github.com/smartdec/smartcheck.

[58] ***Oyente***. [Online]. Available: https://oyente.tech/.

[59] **Securify2**, "ETH-Sri/securify2: Securify v2.0," *GitHub*. [Online]. Available: https://github.com/eth-sri/securify2.

[60] ***Mythril*** , "What is Mythril?¶," *What is Mythril? - Mythril v0.22.24 documentation*. [Online]. Available: https://mythril-classic.readthedocs.io/en/master/about.html.

[61] **Ghaleb, Asem & Pattabiraman, Karthik.** (2020). How effective are smart contract analysis tools? evaluating smart contract static analysis tools using bug injection. 415-427. 10.1145/3395363.3397385.

[62] **Christakis M.,** "Harvey: A Greybox Fuzzer for smart contracts," *arXiv.org*, 15-May-2019. [Online]. Available: https://arxiv.org/abs/1905.06944.

[63] **Christakis** M., "Learning inputs in Greybox fuzzing," *arXiv.org*, 20-Jul-2018. [Online]. Available: https://arxiv.org/abs/1807.07875.

[64] **Crytic**, "Crytic/echidna: Ethereum Smart contract fuzzer," *GitHub*. [Online]. Available: https://github.com/crytic/echidna.

[65] **Kolluri A., Sergey I., Saxena P., and Hobor A.,** "Finding the greedy, prodigal, and suicidal contracts at scale," *arXiv.org*, 14-Mar-2018. [Online]. Available: https://arxiv.org/abs/1802.06038.

[66] **Jiang B., Liu Y., and Chan W. K.**, "Contractfuzzer: Fuzzing smart contracts for Vulnerability Detection," *arXiv.org*, 03-Aug-2018. [Online]. Available: https://arxiv.org/abs/1807.03932.

[67] **Tolmach P., Li Y., Lin S.-W., Liu Y., and Li Z.,** "A survey of Smart Contract Formal Specification and Verification," *arXiv.org*, 19-Apr-2021. [Online]. Available: https://arxiv.org/abs/2008.02712.

[68] **Li, A., Choi, J.A., & Long, F. (2020).** Securing smart contract with runtime validation. *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*.

[69]***Introduction to formal verification*.** [Online]. Available: https://ptolemy.berkeley.edu/projects/embedded/research/vis/doc/VisUser/vis_user/node4.html.

[70] **Li X., Shi Z., Zhang Q., Wang G., Guan Y., Han N.** (2019) Towards Verifying Ethereum Smart Contracts at Intermediate Language Level. In: Ait-Ameur Y., Qin S. (eds) Formal Methods and Software Engineering. ICFEM 2019. Lecture Notes in Computer Science, vol 11852. Springer, Cham. https://doi.org/10.1007/978-3-030-32409-4_8

[71] **Baldoni R, Coppa E., Cono D'elia D., Demetrescu C., and Finocchi I.,** 2018. A Survey of Symbolic Execution Techniques. ACM Comput. Surv. 51, 3, Article 50 (May 2019), 39 pages. DOI:https://doi.org/10.1145/3182657

[72] **CoinMarketCap, CoinMatketCap.** "Rug Pull: CoinMarketCap." CoinMarketCap Alexandria. CoinMarketCap, December 1, 2021. https://coinmarketcap.com/alexandria/glossary/rug-pull.

[73] **Techrate**. "Smart-Contract-Audits/Levelup Full Smart Contract Security ..." Accessed 2021. https://github.com/TechRate/Smart-Contract-Audits/blob/main/StableMagnet%20Standart%20Smart%20Contract%20Security%20Audit.pdf.

[74] **Harvest-Finance**. "Harvest/Audits at Master · Harvest-Finance/Harvest." GitHub. Accessed October 8, 2021. https://github.com/harvest-finance/harvest/tree/master/audits.

[75] **Grimm Finance.** "Smart Contract Audit Services: Solidity Finance." Solidity Finance - Smart Contract Audit Services. Accessed 2021. https://solidity.finance/audits/GrimVaultV2/.

[76] **Deeter**, "Derisking Defi: Guarded Launches." Medium. Electric Capital, May 26, 2020. https://medium.com/electric-capital/derisking-defi-guarded-launches-2600ce730e0a.

[77] **Truffle Suite**. “Truffle Suite.” Truffle Suite - Truffle Suite. Accessed 2021. https://trufflesuite.com/tutorial/index.html.

[78] **Hardhat**. “Ethereum Development Environment for Professionals by NOMIC Labs.” Hardhat. Accessed May 19, 2021. https://hardhat.org/.

[79 **Brownie** , “Brownie”. Accessed June 15, 2021. https://eth-brownie.readthedocs.io/en/stable/.

EKLER

EK 1: Denetim Raporu Taslağı

EK 1: DENETİM RAPORU TASLAĞI

<PROJENİN ADI> DENETİM RAPORU

Hazırlayan: xxxxxx

Başlangıç Tarihi: GG.AA.YYYY

İçindekiler

İçindekiler Tablosu

1. Yönetici Özeti

1.1. Giriş

TODO

1.2. Kapsam

<Denetim kapsamındaki kontratların isimleri buraya>

1.3. Zaman Çizelgesi

<Denetim Sürecinin Zamansal Dağılımı>

1.4. Amaç

Proje hakkında genel bilgi ve projenin denetimi sonucunda neye ulaşılmak istenildiği

1.5. Kullanılan Araçlar ve Yöntemler

Analiz Sürecinde Kullanılan araçlar ve yöntemler

1.6. Projenin Genel Bilgileri

Bu madde projedeki rolleri, yetkileri, varlıkları içerir.

1.7. Bulguların Genel Özeti

Sınıflandırılmış bulguların genel özeti ve bulgu istatistikleri. Burada tablo kullanımı anlaşılırlık açısından daha iyidir

- Risk Skalası:

- 3-5: Projedeki Ehter'in kaybına veya erişilmez hale gelmesine, projenin sahipliğinin değişmesine yol açan zafiyetler
- 2-3: Projenin normal akışının dışında davranışlar göstermesine yol açan ancak Ether kaybına yol açmayan zafiyetler.
- 1-2: Yukarıda anlatılanların dışında kalan türdeki zafiyetler.

- Olasılık Skalası:
  - 3-5: Eğer zafiyet erişim fazla kaynak gerektirmeden birkaç işlemde tetiklenebiliyorsa
  - 2-3: Eğer zafiyet erişim izni, uygun rol, sistemin çalışması hakkında detaylı bilgi ve kaynak gerektiriyorsa
  - 1-2: Yukarıda anlatılanlardan daha zor şartlar ile tetiklenebilen bir zafiyet

Sınıflandırma Skalası:

- 1-3: Bilgilendirme
- 3-6: Düşük
- 6-8: Orta
- 8-10: Yüksek

Tablo 1: Bulguların ciddiyet dağılımı

Yüksek	Orta	Düşük	Bilgilendirme
Sayı	Sayı	Sayı	Sayı

Tablo2: Bulgulara Genel Bakış

Bulgu	Ciddiyet	Çözüm Zamanı
Bulgu 1	Yüksek	Durum: GG.AA.YYYY
Bulgu 2	Orta	Durum: GG.AA.YYYY

2. Bulgular

2.1. Bulgu1 – YÜKSEK

Tanım

Bugu Tanımı ve detayları

Risk Durumu

Olasılık – 5

Ciddiyet - 5

Referanslar

- Referans alına kaynaklar

Öneriler

Zafiyetin Giderilmesi için Çözüm Önerileri

3. Sömürü Senaryoları

Olan Sömürü Senaryolarının detaylı açıklaması (ekran görüntüleri, test ortamı bilgileri vs. gibi detaylar unutulmamalı).