Rapor 7 - Kod Kalitesi ve Profesyonel Altyapı İyileştirmeleri

1. Yönetici Özeti

Bu rapor, eBPF tabanlı fidye yazılımı tespit projesinin "Kavram Kanıtlama" aşamasından, "Sürdürülebilir Mühendislik Ürünü" seviyesine taşınması sürecinde gerçekleştirilen mimari iyileştirmeleri belgelemektedir. Önceki raporlarda (Rapor 1-6) kurulan temel işlevsellik, bu dönemde yapılan çalışmalarla daha kararlı, modüler ve operasyonel olarak yönetilebilir bir yapıya kavuşturulmuştur.

Gerçekleştirilen çalışmalar dört ana başlık altında toplanmaktadır:

1. **Bellek Yönetimi:** EVENT_EXIT kancası ile bellek sızıntılarının önlenmesi.
2. **Modülerleştirme:** Tek parça (monolitik) kod yapısının sorumluluk bazlı katmanlara ayrılması.
3. **Konfigürasyon Yönetimi:** Sabit (hardcoded) değerlerin ransom.conf dosyasına taşınması.
4. **Gelişmiş Loglama:** Olayların ve alarmların kalıcı ve zaman damgalı olarak kayıt altına alınması.

2. Bellek Yönetimi ve Kararlılık (Memory Leak Prevention)

2.1. Problem Tanımı

Sistemin önceki versiyonlarında, her yeni süreç (Process ID - PID) başlatıldığında, durum yönetimi modülü (uthash) tarafından bellekten yer ayrılmaktaydı (malloc). Ancak süreç sonlandığında bu belleğin iade edilmesini (free) sağlayan bir mekanizma bulunmamaktaydı. Bu durum, uzun süreli çalışmalarda RAM tüketiminin sürekli artmasına ve sistemin kararsızlaşmasına yol açma riski taşımaktaydı.

2.2. Teknik Çözüm: EVENT_EXIT Mekanizması

Bu sorunu çözmek için çekirdek ve kullanıcı alanı arasında yeni bir iletişim kanalı kurulmuştur:

- **Çekirdek Tarafı (hello_kern.c):** Linux çekirdeğindeki tracepoint/sched/sched_process_exit izleme noktasına bir eBPF kancası eklenmiştir. Bu kanca, bir süreç sonlandığında kullanıcı alanına sadece PID bilgisini içeren EVENT_EXIT sinyalini gönderir.
- **Kullanıcı Tarafı (state_manager.c):** Gelen EVENT_EXIT sinyali yakalanarak, ilgili PID'ye ait process_stats veri yapısı uthash tablosundan çıkarılır (HASH_DEL) ve bellekten temizlenir (free).

Bu geliştirme, ajanın bellek ayak izinin sabit kalmasını ve sunucu ortamlarında günlerce kesintisiz çalışabilmesini garanti altına almıştır.

2.3. İmplementasyon Detayları

Bu çözümün hayata geçirilmesi için çekirdek alanı, ortak protokol ve kullanıcı alanı katmanlarında aşağıdaki kod değişiklikleri yapılmıştır:

A. Çekirdek Alanı (hello_kern.c)

Linux çekirdeğinin süreç zamanlayıcısındaki sched_process_exit noktasına kanca atılmıştır. Bu nokta, bir process veya thread sonlandığında tetiklenir.

```
// hello_kern.c
// 6. EXIT (Süreç Sonlanması)
// Bir süreç öldüğünde tetiklenir. Bellek temizliği için kritiktir.
SEC("tracepoint/sched/sched_process_exit")
int handle_exit(void *ctx)
{
    // send_event fonksiyonu o anki PID'yi otomatik alır.
    // Dosya adı exit olayında önemsizdir (NULL gönderilir).
    send_event(ctx, EVENT_EXIT, NULL);
    return 0;
}
```

B. Protokol Tanımı (common.h)

Kullanıcı alanının bu yeni olay türünü tanıması için EVENT_EXIT sabiti protokole eklenmiştir.

```
/* common.h */
enum event_type {
    EVENT_EXEC = 1,
    EVENT_WRITE = 2,
    EVENT_OPEN = 3,
    EVENT_RENAME = 4,
    EVENT_EXIT = 5     // <--- YENİ: Süreç Sonlandırma Olayı
};
```

C. Kullanıcı Alanı Temizliği (state_manager.c)

main.c içerisinde gelen olay EVENT_EXIT ise, state_manager modülü tetiklenir ve uthash tablosundan ilgili kayıt silinerek bellek işletim sistemine iade edilir.

```
// state_manager.c
```

```
void remove_process(int pid) {
    struct process_stats *s;
    // Hash tablosunda bu PID var mı?
    HASH_FIND_INT(processes, &pid, s);
    if (s) {
        // Varsa sil ve belleği serbest bırak (free)
        LOG_DEBUG("PID: %d temizleniyor.", pid);
        HASH_DEL(processes, s);
        free(s);
    }
}
```

Bu yapı sayesinde, sistemde binlerce kısa ömürlü süreç (örneğin derleme sırasında oluşan gcc süreçleri) çalışsa bile, ajan sadece aktif olanları hafızasında tutar. Pasif süreçler anında temizlenir.

3. Yazılım Mimarisi: Modüler Dönüşüm

3.1. Monolitik Yapıdan Ayrışma ve Dosya İletişimi

Projenin ilk versiyonlarında, tüm kullanıcı alanı mantığı tek bir C dosyasında (hello_user.c) toplanmıştı. Bu yapı, projenin büyümesiyle birlikte kodun okunabilirliğini, bakımını ve test edilebilirliğini zorlaştırmaktaydı. Yapılan geliştirme ile "Sorumlulukların Ayrılığı" (Separation of Concerns) ilkesi uygulanarak proje, her biri belirli bir göreve odaklanmış modüllere ayrılmıştır.

Aşağıda, sisteme eklenen yeni .c ve .h dosyaları ve bunların birbirleriyle olan iletişim mimarisi detaylandırılmıştır:

3.2. Katmanlı Dosya Yapısı ve Görevleri

A. Çekirdek Alanı

- **hello_kern.c (Veri Toplayıcı):** Sistemdeki olayların kaynağıdır. execve, write, openat, renameat ve exit sistem çağrılarını izler. Kullanıcı alanındaki kütüphanelerden izole çalışır.

B. Ortak Protokol Katmanı

- **common.h:** Çekirdek ve kullanıcı alanı arasındaki "sözleşme"dir. Standart kütüphane bağımlılığı içermez. Her iki tarafın da anladığı veri yapılarını ve olay tiplerini tanımlar.

C. Kullanıcı Alanı Modülleri

1. **main.c (Orkestrasyon ve Giriş Noktası):**
    - **Görevi:** Uygulamanın beynidir. eBPF programını yükler, sonsuz döngüde Ring Buffer'ı dinler ve diğer modülleri koordine eder.
    - **İletişim:** Olay geldiğinde state_manager'dan süreci ister, detector'a analiz ettirir.

2. **state_manager.c / state_manager.h (Durum Yönetimi):**
    - **Görevi:** Sistemin hafızasıdır. Süreçlerin (PID) istatistiklerini (toplam yazma, başlangıç zamanı vb.) uthash tablosunda tutar.
    - **Özelliği:** Bellek yönetimi (malloc/free) burada yapılır. main.c tarafından çağrılır.
3. **detector.c / detector.h (Analiz Motoru):**
    - **Görevi:** Sistemin karar mekanizmasıdır. Sadece matematiksel analiz yapar.
    - **İşleyişi:** Bir olayın anomali olup olmadığını belirlemek için süreç geçmişini ve config modülünden gelen eşik değerlerini kullanır. Şu an H1 (Hız Analizi) kuralını barındırır.
4. **config.c / config.h (Konfigürasyon Yönetimi):**
    - **Görevi:** Sistemin ayar deposudur. Başlangıçta ransom.conf dosyasını okuyarak çalışma parametrelerini (struct app_config) belirler.
    - **İletişim:** detector ve logger modülleri, eşik değerlerine ve dosya yollarına bu modül üzerinden erişir.
5. **logger.c / logger.h (Loglama Altyapısı):**
    - **Görevi:** Sistemin sesi ve kayıt defteridir.
    - **İşleyişi:** Hibrit bir yapı sunar; geliştirici modunda terminale renkli çıktı basarken, operasyon modunda olayları /var/log altındaki dosyaya zaman damgalı olarak kaydeder.

3.3. Modüller Arası Veri Akış Senaryosu

Bir "Dosya Şifreleme Girişimi" sırasında sistem içindeki veri akışı şu şekildedir:

1. **Çekirdek:** hello_kern.c, şifreleme (write) işlemini yakalar ve common.h yapısına paketleyip Ring Buffer'a atar.
2. **Okuma:** main.c veriyi okur ve config.c ayarlarını kontrol eder.
3. **Hafıza:** main.c, state_manager.c'ye sorar: "Bu PID'nin geçmişi var mı?". Varsa getirir, yoksa oluşturur.
4. **Analiz:** main.c, olayı ve süreç geçmişini detector.c'ye gönderir. detector.c, config.c'deki eşik değerine bakarak (Örn: >15 yazma/sn) kontrol eder.
5. **Alarm:** Eşik aşılmışsa, detector.c doğrudan logger.c modülünü çağırarak "Kritik Alarm" üretir.
6. **Kayıt:** logger.c bu alarmı hem ekrana basar hem de log dosyasına yazar.

4. Konfigürasyon Yönetimi

4.1. Statik Değerlerden Dinamik Yapılandırmaya

Önceki versiyonlarda, analiz penceresi (RATE_WINDOW_SEC) ve eşik değerleri (THRESHOLD_WRITE) kaynak kodun (detector.h) içine #define makroları olarak gömülüydü. Bu durum, en ufak bir hassasiyet ayarı veya log dosyası yolu değişikliği için kaynak kodun düzenlenmesini ve projenin yeniden derlenmesini (recompile) gerektiriyordu. Bu yaklaşım, operasyonel esnekliği kısıtlamakta ve "Hardcoded" veri kullanımı nedeniyle kod kalitesini düşürmekteydi.

4.2. Teknik İmplementasyon: config Modülü

Bu kısıtlamayı aşmak için projeye config.c ve config.h dosyalarından oluşan bir konfigürasyon yönetim modülü eklenmiştir.

- **config.h (Veri Yapısı):** Tüm uygulama ayarlarını tek bir çatı altında toplayan struct app_config yapısını tanımlar. Ayrıca dosya okunamadığı durumlar için güvenli varsayılan değerleri (fallback values) barındırır.
- **config.c (Ayrıştırıcı/Parser):** Uygulama başlatıldığında ransom.conf dosyasını satır satır okur. Standart C kütüphanesi fonksiyonlarını (fopen, fgets, sscanf) kullanarak ANAHTAR=DEĞER formatındaki satırları ayrıştırır ve global konfigürasyon nesnesini günceller.

4.3. ransom.conf Entegrasyonu ve Kullanımı

Sisteme eklenen ransom.conf dosyası, son kullanıcının veya sistem yöneticisinin aracı derlemeden yönetebilmesini sağlar. Dosya içeriği aşağıdaki yapıdadır:

```
# eBPF Ransomware Detection Config
# Format: ANAHTAR=DEGER

# Analiz penceresi (saniye)
WINDOW_SEC=5

# Yazma hızı limiti (dosya/pencere)
WRITE_THRESHOLD=100

# Yeniden adlandırma limiti
RENAME_THRESHOLD=10

# Log dosyası konumu
LOG_FILE=/var/log/ransom-bpf.log
```

Uygulama başladığında (main.c), bu dosyayı arar ve yükler. Eğer dosya bulunamazsa, sistem güvenli varsayılan değerlerle çalışmaya devam eder ve logger modülü üzerinden bir uyarı (WARN) mesajı basar. Bu yapı, aracın farklı ortamlara (Geliştirme, Test, Prodüksiyon) kod değişikliği yapılmadan kolayca uyarlanabilmesini sağlamaktadır.

5. Gelişmiş Loglama ve Adli Analiz (Forensics)

5.1. Kalıcılık İhtiyacı

Siber güvenlik olaylarında, saldırı anının tespiti kadar saldırı sonrası analiz (post-incident analysis) de kritiktir. Yalnızca terminal ekranına (stdout) basılan loglar uçucudur ve olay sonrası inceleme için yetersizdir.

5.2. Hibrit Loglama Yapısı

Geliştirilen logger.c modülü ile sistem artık "Hibrit Loglama" yeteneğine sahiptir. Bu yapı, hem geliştirme hem de canlı kullanım (production) senaryolarını desteklemek üzere tasarlanmıştır.

1. **Terminal (Geliştirici Modu):** stdout ve stderr kanallarını kullanarak olayları gerçek zamanlı ve **renkli** bir formatta ekrana basar. Bu, geliştirme sırasında hata ayıklamayı kolaylaştırır.
    - [INFO] -> Yeşil
    - [ALARM] -> Kırmızı
    - [WARN] -> Sarı
2. **Dosya (Operasyon Modu):** Tüm olaylar ve alarmlar, ransom.conf dosyasında belirtilen konuma (varsayılan: /var/log/ransom-bpf.log), **zaman damgalı** ve düz metin olarak kaydedilir.

5.3. Adli Analiz Yeteneği

Log dosyasına eklenen zaman damgası (YYYY-MM-DD HH:MM:SS), olayların kronolojik sırasını netleştirir. Bu sayede, bir saldırı durumunda aşağıdaki sorulara yanıt bulunabilir:

- Saldırgan hangi saatte sisteme giriş yaptı (SSH Login)?
- İlk şifreleme girişimi ne zaman başladı?
- Saldırgan hangi dosyaları hedef aldı?

Örnek Log Kaydı:
[2025-10-26 14:30:05] [ALARM] FIDYE YAZILIMI SUPHESI (WRITE BURST)! PID: 4521
Bu yapı, geçmişe dönük saldırı analizi ve sistem denetimi (audit) için gerekli temel altyapıyı sağlar.

6. Sonuç

Bu rapor döneminde yapılan çalışmalarla, eBPF projesi teknik bir prototipten, ölçeklenebilir ve yönetilebilir bir güvenlik yazılımına evrilmiştir.

- **Kararlılık:** Bellek sızıntısı giderildi.
- **Esneklik:** Konfigürasyon dosyası desteği eklendi.
- **İzlenebilirlik:** Kalıcı loglama altyapısı kuruldu.
- **Sürdürülebilirlik:** Kod tabanı modüler hale getirildi.

Proje, bu sağlam temel üzerinde daha karmaşık tespit algoritmalarını (H2 - İmza Bazlı Tespit) barındırmaya hazırdır.