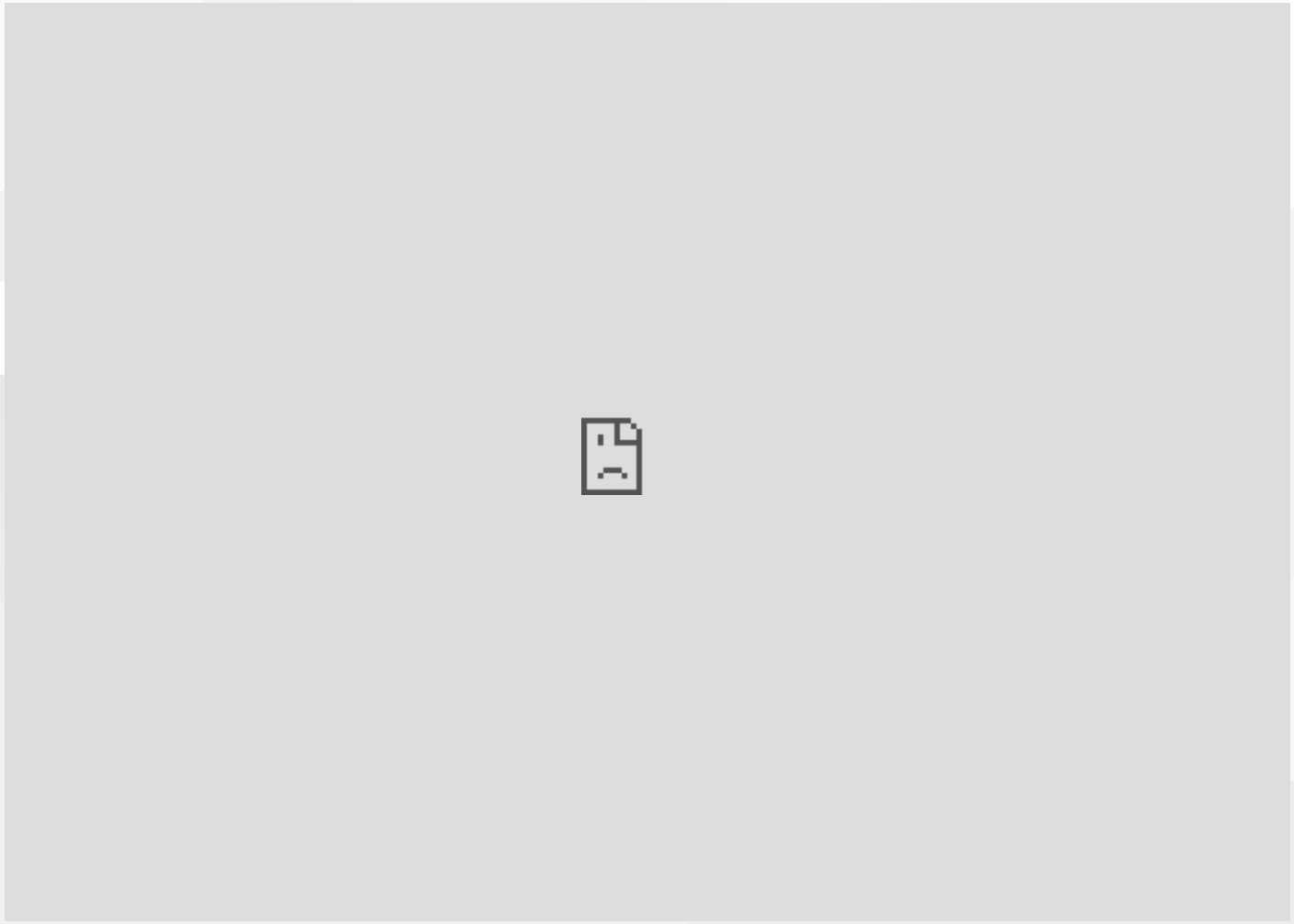
CEN429 Güvenli Programlama

Hafta-4

Kod Güçlendirme Teknikleri

İndir

- PDF
- DOC
- SLIDE
- PPTX



Outline

- Kod Güçlendirme Teknikleri
- Native C/C++ İçin Kod Güçlendirme
- Java ve Yorumlanan Diller İçin Kod Güçlendirme

Hafta-4: Kod Güçlendirme Teknikleri

1. Native C/C++ İçin Kod Güçlendirme Teknikleri

C ve C++ gibi düşük seviye dillerde güvenli kod yazmak ve saldırılara karşı dayanıklı hale getirmek için çeşitli teknikler kullanılır. Bu teknikler, kodun analiz edilmesini ve geri mühendislik işlemlerini zorlaştırmayı amaçlar.

1.1 Opaque Loops (Opak Döngüler)

Teorik Açıklama: Opak döngüler, dışarıdan bakıldığında amacı belli olmayan döngülerdir. Bu döngüler sayesinde kodun analizi zorlaşır. Saldırgan, döngünün işlevini anlamakta zorlanır ve kodun çözülmesi daha karmaşık hale gelir.

Uygulama Örnekleri:

1. Rastgele bir koşul ile oluşturulmuş döngüler ekleyerek kodun analizini zorlaştırma.
2. Dışarıdan anlaşılmayan ancak programın işleyişine zarar vermeyen döngüler ekleme.
3. Opak döngüler ile programın çalışma süresini arttırarak saldırganı yanıltma.

1.1.1. Rastgele Bir Koşul ile Oluşturulmuş Döngü (Opaque Loop with Random Condition)

Bu örnekte, rastgele bir koşul ile döngü oluşturularak kodun anlaşılması zorlaştırılıyor.

```
#include <iostream>
#include <cstdlib>
#include <ctime>

void opak_rastgele_dongu() {
    srand(time(0)); // Rastgele sayı üreticisini başlatır
    int x = rand() % 50; // Döngü koşulu için rastgele bir değer
    for (int i = 0; i < x; i++) {
        if (i % 2 == 0) {
            std::cout << "İterasyon: " << i << std::endl;
        }
    }
}
```

```
int main() {
    opak_rastgele_dongu();
    return 0;
}
```

Özet:

Bu örnekte, döngü koşulu rastgele bir sayıdan oluşur. Bu durum, döngünün amacını dışarıdan anlamayı zorlaştırır ve saldırganlar için analizi karmaşık hale getirir.

1.1.2. Dışarıdan Anlaşılmayan Ancak Programın İşleyişine Zarar Vermeyen Döngü (Opaque Loop with No External Effect)

Bu örnekte, döngü kodun işleyişine zarar vermeyen ancak dışarıdan anlaşılmayan bir işlev içerir.

```cpp
#include <iostream>

void opak_islevsiz_dongu() {
    for (int i = 0; i < 100; i++) {
        if (i % 3 == 0) {
            int gizli_islem = i * 42; // İşleyişe etkisi olmayan gereksiz işlem
        }
    }
    std::cout << "Opak döngü tamamlandı." << std::endl;
}
```

```c
int main() {
    opak_islevsiz_dongu();
    return 0;
}
```

Özet:

Bu örnekte, döngüde yapılan işlem programın ana işleyişine katkı sağlamaz. Bu tür döngüler, saldırganları yanıltmak ve kodun analizini zorlaştırmak için kullanılır.

1.1.3. Opak Döngüler ile Programın Çalışma Süresini Arttırma (Opaque Loops to Delay Program Execution)

Bu örnek, programın çalışma süresini uzatarak saldırganları yanıltmak amacıyla kullanılabilir.

```
#include <iostream>
#include <thread>
#include <chrono>

void gecikmeli_opak_dongu() {
    for (int i = 0; i < 5; i++) {
        std::this_thread::sleep_for(std::chrono::seconds(1)); // Her döngüde 1 saniye bekleme
        std::cout << "Gecikmeli döngü iterasyonu: " << i << std::endl;
    }
}
```

```
int main() {
    gecikmeli_opak_dongu();
    return 0;
}
```

Özet:

Bu örnekte, döngü her iterasyonda programın çalışmasını yavaşlatan bir gecikme ekler. Bu tür teknikler, saldırganların programın tam olarak ne yaptığı konusunda kafa karışıklığı yaratmak için kullanılır.

1.2 Shared Object Sembollerini Gizleme (Configure Shared Object Symbol Invisible)

Teorik Açıklama: Paylaşılan nesneler (shared object) içinde kullanılan sembollerin gizlenmesi, bu nesnelere dışarıdan erişimi zorlaştırır. Bu işlem, analiz ve geri mühendislik işlemlerini engellemek için kullanılır.

Uygulama Örnekleri:

1. Derleyici seçenekleriyle sembollerin görünürlüğünü sınırlama.
2. Sadece gerekli sembolleri dışa açarak diğer sembollerin erişilemez olmasını sağlama.
3. Paylaşılan kütüphanelerdeki kritik fonksiyonları gizleyerek güvenliği artırma.

1.2.1. Derleyici Seçenekleriyle Sembollerin Görünürlüğünü Sınırlama (Limiting Symbol Visibility with Compiler Options)

Açıklama:

Paylaşılan nesne dosyalarında, sembollerin varsayılan olarak dışa açık (public) olması, güvenlik açıklarına yol açabilir. Bu durumu önlemek için derleyici seçenekleriyle sembollerin görünürlüğü sınırlandırılabilir. Örneğin, GCC ve Clang derleyicilerinde `-fvisibility=hidden` seçeneği kullanılarak, sadece dışa açılmasına izin verilen semboller görünür olur.

Örnek:

```
// foo.cpp
#include <iostream>

void gizli_fonksiyon() __attribute__((visibility("hidden"))); // Fonksiyon gizli

void gizli_fonksiyon() {
    std::cout << "Bu fonksiyon dışa açık değildir." << std::endl;
}

void acik_fonksiyon() {
    std::cout << "Bu fonksiyon dışa açıktır." << std::endl;
}
```

Derleme Komutu:

```
g++ -fvisibility=hidden -shared -o libfoo.so foo.cpp
```

Özet:

Bu komut, tüm sembolleri varsayılan olarak gizler (`-fvisibility=hidden`), ancak `acik_fonksiyon` dışa açılabilir durumda kalır. `gizli_fonksiyon` ise dışarıdan erişilemez olur.

1.2.2. Sadece Gerekli Sembolleri Dışa Açma (Only Exporting Necessary Symbols)

Açıklama:

Paylaşılan kütüphanelerde yalnızca gerekli semboller dışa açılır. Bu sayede, sadece belirli fonksiyonlar dışarıdan çağrılabilirken diğer semboller gizli kalır.

Örnek:

```
// gizli_kutuphane.cpp
#include <iostream>

void gizli_fonksiyon() {
    std::cout << "Bu fonksiyon gizli kalacak." << std::endl;
}

extern "C" void acik_fonksiyon() {
    std::cout << "Bu fonksiyon dışa açık." << std::endl;
}
```

Derleme Komutu:

```
g++ -fvisibility=hidden -shared -o libgizli.so gizli_kutuphane.cpp
```

Özet:

Bu örnekte, `acik_fonksiyon` dışa açılabilirken, `gizli_fonksiyon` dışarıdan erişilemez durumda kalır. Bu teknik, sadece dışarıdan kullanılmasına izin verilen fonksiyonların erişime açılmasını sağlar.

1.2.3. Paylaşılan Kütüphanelerde Kritik Fonksiyonları Gizleme (Hiding Critical Functions in Shared Libraries)

Açıklama:

Kritik öneme sahip fonksiyonlar gizlenerek, geri mühendislik işlemlerini zorlaştırmak ve paylaşılan kütüphanelerin güvenliğini artırmak mümkündür. Bu yaklaşım, saldırganların önemli fonksiyonları analiz etmesini ve manipüle etmesini önler.

Örnek:

```
// kritik_kutuphane.cpp
#include <iostream>

__attribute__((visibility("hidden"))) void kritik_fonksiyon() {
    std::cout << "Bu kritik fonksiyon gizlenmiştir." << std::endl;
}

extern "C" void genel_fonksiyon() {
    std::cout << "Bu genel fonksiyon dışa açıktır." << std::endl;
    kritik_fonksiyon(); // Gizli fonksiyon burada içsel olarak çağrılır
}
```

Derleme Komutu:

```
g++ -fvisibility=hidden -shared -o libkritik.so kritik_kutuphane.cpp
```

Özet:

Bu örnekte, `kritik_fonksiyon` dışarıdan erişilemez ve yalnızca `genel_fonksiyon` üzerinden çağrılabilir. Bu, kritik fonksiyonların dışa kapalı kalmasını sağlayarak güvenliği artırır.

1.3. Aritmetik İşlemlerin Obfuske Edilmesi (Obfuscation of Arithmetic Instructions)

Teorik Açıklama: Aritmetik işlemler, programın en temel yapı taşlarıdır. Bu işlemleri karmaşık hale getirmek, kodun analizini ve anlaşılmasını zorlaştırır.

Uygulama Örnekleri:

1. Basit toplama işlemlerini daha karmaşık matematiksel ifadeler ile değiştirme.
2. Aritmetik işlemlerine gereksiz adımlar ekleyerek işlevselliği korurken kodun anlaşılmasını zorlaştırma.
3. Aritmetik işlemler üzerinde bit manipülasyonu yaparak daha karmaşık hale getirme.

1.3.1. Basit Toplama İşlemlerini Daha Karmaşık Matematiksel İfadeler ile Değiştirme (Replacing Simple Additions with Complex Mathematical Expressions)

Açıklama:

Basit aritmetik işlemlerini karmaşık hale getirerek, kodun anlaşılmasını zorlaştırabiliriz.

Örneğin, bir toplama işlemini daha uzun ve karmaşık matematiksel işlemlerle değiştirmek, saldırganların kodu analiz etmesini zorlaştırır.

Örnek:

```
#include <iostream>

int karmaşık_toplama(int a, int b) {
    // Basit toplama işlemi: a + b
    // Karmaşık hale getirilmiş hali
    return ((a * 2) + (b * 2) - a - b); // a + b ile aynı sonucu verir
}
```

```
int main() {
    int a = 5, b = 10;
    int sonuc = karmaşık_toplama(a, b);
    std::cout << "Karmaşık toplama sonucu: " << sonuc << std::endl;
    return 0;
}
```

Özet:

Bu kodda basit bir toplama işlemi (a + b) daha karmaşık bir matematiksel ifadeye dönüştürülmüştür. Her ne kadar sonuç aynı olsa da kodun analizi zorlaşır.

1.3.2. Aritmetik İşlemlerine Gereksiz Adımlar Ekleyerek Kodun Anlaşılmasını Zorlaştırma (Adding Redundant Steps to Obfuscate Arithmetic Operations)

Açıklama:

Aritmetik işlemlerine gereksiz adımlar eklemek, işlevi değiştirmeden kodun anlaşılmasını zorlaştırır. Örneğin, ekleme ve çıkarma işlemleri eklenerek kod daha karmaşık hale getirilebilir.

Örnek:

```
#include <iostream>

int gereksiz_adimlarla_toplama(int a, int b) {
    // Toplama işlemini gereksiz adımlarla karmaşıklaştırma
    int sonuc = (a * 2 - a) + (b * 2 - b); // a + b işlemini yapar
    return sonuc;
}
```

```
int main() {
    int a = 7, b = 3;
    int sonuc = gereksiz_adimlarla_toplama(a, b);
    std::cout << "Gereksiz adımlarla toplama sonucu: " << sonuc << std::endl;
    return 0;
}
```

Özet:

Bu örnekte, toplama işlemi gereksiz adımlarla karmaşık hale getirilmiştir. Sonuç yine `a + b` olsa da işlem, dışarıdan bakıldığında anlaşılması zor hale gelmiştir.

1.3.3. Aritmetik İşlemler Üzerinde Bit Manipülasyonu Yaparak Karmaşık Hale Getirme (Adding Bit Manipulation to Obfuscate Arithmetic Operations)

Açıklama:

Aritmetik işlemlere bit manipülasyonu ekleyerek kodu daha da karmaşık hale getirebiliriz. Bu yöntem, özellikle düşük seviyeli dillerde kodun geri mühendislik işlemine karşı dayanıklılığını artırır.

Örnek:

```
#include <iostream>

int bit_manipulasyonu_ile_toplama(int a, int b) {
    // Aritmetik işlemi bit manipülasyonu ile karmaşık hale getirme
    int sonuc = ((a << 1) >> 1) + ((b << 1) >> 1); // a + b işlemi yapar
    return sonuc;
}
```

```
int main() {
    int a = 4, b = 8;
    int sonuc = bit_manipulasyonu_ile_toplama(a, b);
    std::cout << "Bit manipülasyonu ile toplama sonucu: " << sonuc << std::endl;
    return 0;
}
```

Özet:

Bu örnekte, toplama işlemi bit kaydırma işlemleri (left shift, right shift) ile karmaşık hale getirilmiştir. Bit manipülasyonu, aritmetik işlemi gizler ve kodun analiz edilmesini zorlaştırır.

1.4. Fonksiyon İsimlerinin Obfuske Edilmesi (Obfuscation of Function Names)

Teorik Açıklama: Fonksiyon isimlerinin rastgele karakter dizileri ile değiştirilmesi, kodun anlaşılmasını zorlaştırır. Bu teknik, özellikle tersine mühendislik (reverse engineering) işlemlerini engellemek için kullanılır.

Uygulama Örnekleri:

1. Fonksiyon isimlerini anlamsız karakter dizileri ile değiştirme.
2. Her derlemede farklı fonksiyon isimleri oluşturarak statik analiz araçlarını yanıltma.
3. Kritik fonksiyonların isimlerini rastgele hale getirerek saldırganların bu fonksiyonları anlamasını zorlaştırma.

1.4.1. Fonksiyon İsimlerini Anlamsız Karakter Dizileri ile Değiştirme (Replacing Function Names with Nonsense Character Strings)

Açıklama:

Fonksiyon isimleri, anlaşılmayı zorlaştırmak amacıyla anlamsız karakter dizileri ile değiştirilir. Bu yöntem, saldırganların hangi fonksiyonun ne işe yaradığını anlamasını zorlaştırır.

Örnek:

```
#include <iostream>

// Fonksiyon isimleri anlamsız karakter dizileriyle değiştirilmiş
void abcdef123() {
    std::cout << "Kritik bir işlem gerçekleştirildi." << std::endl;
}
```

```
int main() {
    abcdef123(); // Fonksiyon çağrısı
    return 0;
}
```

Özet:

Bu örnekte, `abcdef123` gibi anlamsız bir karakter dizisi kullanılarak fonksiyon ismi gizlenmiştir. Bu durum, kodun anlaşılmasını ve analiz edilmesini zorlaştırır.

1.4.2. Her Derlemede Farklı Fonksiyon İsimleri Oluşturarak Statik Analiz Araçlarını Yanıltma (Generating Different Function Names on Every Compile)

Açıklama:

Her derlemede farklı fonksiyon isimleri oluşturmak, statik analiz araçlarını ve geri mühendislik işlemlerini zorlaştırabilir. Derleme sırasında fonksiyon isimleri rastgele oluşturularak kod analizi karmaşık hale getirilir.

Örnek:

```
#include <iostream>
#include <cstdlib>
#include <ctime>

// Rastgele fonksiyon ismi oluşturma
#define OBFUSCATED_FUNC_NAME random_function_name

void OBFUSCATED_FUNC_NAME() {
    std::cout << "Bu fonksiyonun ismi her derlemede değişebilir." << std::endl;
}
```

```c
int main() {
    OBFUSCATED_FUNC_NAME(); // Fonksiyon çağrısı
    return 0;
}
```

Özet:

Makro tanımı ve derleme sürecinde rastgele fonksiyon isimleri oluşturmak, her derlemede farklı bir isim kullanarak kodun statik analiz araçları tarafından analiz edilmesini zorlaştırır.

1.4.3. Kritik Fonksiyonların İsimlerini Rastgele Hale Getirerek Saldırganların Bu Fonksiyonları Anlamasını Zorlaştırma (Randomizing Critical Function Names to Obfuscate Purpose)

Açıklama:

Kritik fonksiyonların isimlerini rastgele hale getirmek, saldırganların bu fonksiyonların amacını anlamasını zorlaştırır. Bu teknik, özellikle geri mühendislik işlemlerini engellemek için kullanılır.

Örnek:

```
#include <iostream>

// Kritik fonksiyonların isimleri rastgele belirlenmiş
void xj239rf84() {
    std::cout << "Kritik bir güvenlik işlemi gerçekleştiriliyor." << std::endl;
}

void z8kd93p2() {
    std::cout << "Kritik bir doğrulama işlemi gerçekleştiriliyor." << std::endl;
}
```

```
int main() {
    xj239rf84(); // Kritik güvenlik fonksiyonu çağrısı
    z8kd93p2();  // Kritik doğrulama fonksiyonu çağrısı
    return 0;
}
```

Özet:

Bu örnekte kritik fonksiyonlar rastgele isimler almıştır (`xj239rf84` , `z8kd93p2`). Bu, tersine mühendislik işlemlerini zorlaştırır, çünkü isimler fonksiyonun işlevi hakkında bilgi vermez.

1.5. Kaynak Dosya İsimlerinin Obfuske Edilmesi (Obfuscation of Source File Names)

Teorik Açıklama: Kaynak dosyaların isimlerini anlamsız hale getirerek kodun hangi fonksiyona veya sınıfa ait olduğunu gizleme.

Uygulama Örnekleri:

1. Kaynak dosyaların isimlerini rastgele karakterler ile değiştirme.
2. Kaynak dosyalar arasındaki ilişkiyi gizleyerek kod yapısını anlaşılmaz hale getirme.
3. Dosya isimlerini obfuske ederken kaynak kodu etkilemeyecek şekilde yapıları değiştirme.

1.5.1. Kaynak Dosyaların İsimlerini Rastgele Karakterler ile Değiştirme (Randomizing Source File Names)

Açıklama:

Kaynak dosya isimlerini rastgele karakter dizileri ile değiştirerek, bu dosyaların hangi işlevleri barındırdığını gizleyebiliriz. Bu teknik, saldırganların hangi dosyanın hangi işlemi gerçekleştirdiğini anlamasını zorlaştırır.

Örnek:

1. **Orijinal Dosya Adı:**

```
hesaplama.cpp
```

```
// hesaplama.cpp
#include <iostream>

void hesapla() {
    std::cout << "Hesaplama işlemi" << std::endl;
}
```

2. Obfuske Edilmiş Dosya Adı:

```
x7z23f.cpp
```

```cpp
// x7z23f.cpp
#include <iostream>

void x7z23f() {
    std::cout << "Hesaplama işlemi" << std::endl;
}
```

Özet:

Bu örnekte, `hesaplama.cpp` adlı dosyanın adı `x7z23f.cpp` olarak değiştirilmiştir. Ayrıca, fonksiyon adı da aynı rastgele karakterlerle değiştirilmiştir. Bu, kod yapısını gizlemek için etkili bir yöntemdir.

1.5.2. Kaynak Dosyalar Arasındaki İlişkiyi Gizleyerek Kod Yapısını Anlaşılmaz Hale Getirme (Hiding Relationships Between Source Files)

Açıklama:

Kod yapısındaki dosyalar arasındaki ilişkiyi gizlemek, saldırganların kaynak dosyaların nasıl etkileşimde bulunduğunu anlamasını zorlaştırır. Bu teknik, kodun genel yapısını daha gizli hale getirir.

Örnek:

1. **Orijinal Dosyalar:**
   - `hesaplama.cpp`
   - `utils.cpp`

```cpp
// hesaplama.cpp
#include "utils.h"

void hesapla() {
    int sonuc = toplama(5, 10); // utils.cpp dosyasındaki fonksiyon çağrısı
    std::cout << "Sonuç: " << sonuc << std::endl;
}
```

```cpp
// utils.cpp
int toplama(int a, int b) {
    return a + b;
}
```

Obfuske Edilmiş Dosyalar:

- `a9s8d.cpp`
- `p2f6k.cpp`

```cpp
// a9s8d.cpp
#include "p2f6k.h"

void a9s8d() {
    int sonuc = p2f6k(5, 10); // Fonksiyon ilişkisi gizlenmiş
    std::cout << "Sonuç: " << sonuc << std::endl;
}
```

```cpp
// p2f6k.cpp
int p2f6k(int a, int b) {
    return a + b;
}
```

Özet:

Bu örnekte, iki dosya arasındaki ilişki dosya isimlerinin ve fonksiyon isimlerinin değiştirilmesiyle gizlenmiştir. Dosyalar arasındaki ilişki, dışarıdan bakıldığında anlaşılamaz hale getirilmiştir.

1.5.3. Dosya İsimlerini Obfuske Ederken Kaynak Kodu Etkilemeyecek Şekilde Yapıları Değiştirme (Obfuscating File Names Without Affecting Source Code Structure)

Açıklama:

Dosya isimleri obfuske edilse bile kaynak kodun çalışma mantığı değiştirilmez. Derleme sırasında dosya isimleri ve kod yapıları arasında doğru bağlantı kurularak kodun işlevselliği korunur.

Örnek:

1. **Orijinal Dosya Yapısı:**

```
kaynak/
├── hesaplama.cpp
└── utils.cpp
```

```cpp
// hesaplama.cpp
#include "utils.h"

void hesapla() {
    std::cout << "Hesaplama işlemi başladı." << std::endl;
}
```

2. **Obfuske Edilmiş Dosya Yapısı:**

```
kaynak/
├── q1w2e.cpp
└── z3x4c.cpp
```

```cpp
// q1w2e.cpp
#include "z3x4c.h"

void q1w2e() {
    std::cout << "Hesaplama işlemi başladı." << std::endl;
}
```

Özet:

Dosya isimleri ve fonksiyon isimleri değiştirilmiş olsa da, dosyalar arasındaki ilişki doğru referanslarla korunmuş ve kaynak kodun işlevselliği etkilenmemiştir.

1.6. Statik Dizelerin Obfuske Edilmesi (Obfuscation of Static Strings)

Teorik Açıklama: Statik dizeler, saldırganlar için önemli bilgi kaynaklarıdır. Bu dizelerin şifrelenmesi ve gizlenmesi, kod güvenliğini artırır.

Uygulama Örnekleri:

1. Statik dizeleri şifreleyerek çalışma anında çözülmesini sağlama.
2. Rastgele dize maskeleri uygulayarak dizelerin anlamını gizleme.
3. Dize sabitlerini kaldırarak sabit dize kullanımını azaltma.

1.6.1. Statik Dizeleri Şifreleyerek Çalışma Anında Çözülmesini Sağlama (Encrypting Static Strings and Decrypting at Runtime)

Açıklama:

Statik dizeler saldırganlar için önemli bilgi kaynakları olabilir. Bu yüzden, dizeler çalışma zamanında şifrelenip, yalnızca gerekli olduğunda çözülerek kullanılabilir.

Örnek:

```
#include <iostream>
#include <string>

// Basit bir XOR şifreleme ve çözme fonksiyonu
std::string xor_sifrele(const std::string &input, char key) {
    std::string output = input;
    for (size_t i = 0; i < input.size(); i++) {
        output[i] ^= key; // XOR işlemi
    }
    return output;
}
```

```
int main() {
    std::string sifreli_dize = xor_sifrele("GizliMesaj", 0xAA); // Şifreleme
    std::cout << "Şifrelenmiş Dize: " << sifreli_dize << std::endl;

    std::string cozulmus_dize = xor_sifrele(sifreli_dize, 0xAA); // Çözme
    std::cout << "Çözülmüş Dize: " << cozulmus_dize << std::endl;

    return 0;
}
```

Özet:

Bu örnekte, statik bir dize (`"GizliMesaj"`) XOR işlemi kullanılarak şifrelenmiştir. Dizeler çalışma anında çözülerek anlamlı hale getirilir ve saldırganların dizeleri statik analiz araçlarıyla doğrudan görmesi engellenir.

1.6.2. Rastgele Dize Maskeleri Uygulayarak Dizelerin Anlamını Gizleme (Applying Random String Masks to Obfuscate String Meaning)

Açıklama:

Statik dizeler üzerine rastgele maskeler uygulanarak dizelerin anlamı gizlenir. Bu teknik, saldırganların şifrelenmiş dizeleri çözmesini zorlaştırır.

Örnek:

```
#include <iostream>
#include <string>

std::string dize_maskele(const std::string &input) {
    std::string output = input;
    for (size_t i = 0; i < input.size(); i++) {
        output[i] ^= (i % 255); // Rastgele bir maskeleme işlemi
    }
    return output;
}
```

```
int main() {
    std::string orijinal_dize = "ÖnemliBilgi";
    std::string maske_dize = dize_maskele(orijinal_dize); // Maskeleme
    std::cout << "Masked Dize: " << maske_dize << std::endl;

    std::string cozulmus_dize = dize_maskele(maske_dize); // Maskeleme ters işlemi
    std::cout << "Çözülmüş Dize: " << cozulmus_dize << std::endl;

    return 0;
}
```

Özet:

Bu örnekte, dizenin her karakterine maskeleme uygulanarak dize karmaşık hale getirilmiştir. Çalışma anında maskeler ters çevrilerek dizenin anlamı tekrar ortaya çıkar. Bu yöntem, dizelerin doğrudan okunmasını zorlaştırır.

1.6.3. Dize Sabitlerini Kaldırarak Sabit Dize Kullanımını Azaltma (Reducing the Use of Static String Constants)

Açıklama:

Kodda sabit dizeler kullanmak, saldırganlar için ipuçları sağlayabilir. Bu yüzden sabit dize kullanımını en aza indirerek ve çalışma zamanında dizeleri oluşturarak güvenliği artırmak mümkündür.

Örnek:

```
#include <iostream>
#include <sstream>

std::string dinamik_dize_olustur() {
    std::ostringstream ss;
    ss << "Parola" << "2024"; // Sabit dizeleri dinamik olarak birleştiriyoruz
    return ss.str();
}
```

```
int main() {
    std::string parola = dinamik_dize_olustur(); // Parola dinamik olarak oluşturuluyor
    std::cout << "Dinamik Dize: " << parola << std::endl;

    return 0;
}
```

Özet:

Bu örnekte, sabit dize yerine, dizeler çalışma zamanında dinamik olarak oluşturulmuştur. Bu yaklaşım, kod içinde sabit dizelerin bulunmasını ve bu dizelere saldırı yapılmasını zorlaştırır.

1.7. Opak Boolean Değişkenler (Opaque Boolean Variables)

Teorik Açıklama:

Opak boolean değişkenler, koşullu ifadelerin anlaşılmasını zorlaştırmak için kullanılır. Bu teknik, koşulların karmaşık hale getirilmesiyle kodun analizini ve geri mühendislik işlemlerini güçleştirir.

Örnek Önerisi:

- Rastgele boolean değerleri döndüren bir fonksiyonun koşullu ifadelerde kullanılması.
- Şartların karmaşıklaştırılmasıyla kodun öngörülemez hale getirilmesi.

1.7.1. Rastgele Boolean Değerleri Döndüren Bir Fonksiyonun Kullanılması

Açıklama:

Bu örnekte, rastgele boolean değer döndüren bir fonksiyon, koşullu ifadelerde kullanılarak kodun öngörülemez hale getirilmesi sağlanmıştır. Bu durum, kodun anlaşılmasını ve analiz edilmesini zorlaştırır.

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

// Rastgele opak boolean değer döndüren fonksiyon
bool opak_boolean() {
    srand(time(0)); // Rastgele sayı üretici başlatılıyor
    return rand() % 2; // Rastgele true veya false döner
}
```

```
void gizli_islem(int a) {
    bool durum = opak_boolean(); // Rastgele boolean koşul
    if (a > 10 && durum) {
        std::cout << "Gizli işlem çalıştırılıyor, durum: true" << std::endl;
    } else {
        std::cout << "Koşul sağlanmadı, durum: false" << std::endl;
    }
}
```

```c
int main() {
    gizli_islem(12); // Girdi değeriyle fonksiyon çağrılıyor
    gizli_islem(7);  // Farklı girdi değeriyle tekrar çağrılıyor
    return 0;
}
```

Özet:

Bu kodda, `opak_boolean` fonksiyonu rastgele olarak `true` ya da `false` döndürür. Koşullu ifade, bu rastgele değerle birleştiğinde saldırganlar için kodun analizi zorlaşır.

1.7.2. Şartların Karmaşıklaştırılması ile Kodun Öngörülemez Hale Getirilmesi

Açıklama:

Bu örnek, boolean değerleriyle karmaşık koşullar oluşturularak kodun öngörülemez hale getirilmesini sağlar. Bu tür karmaşık koşullar, kodun geri mühendislik sürecini zorlaştırır.

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

// Karmaşık boolean döndüren fonksiyon
bool karmaşık_boolean(int a) {
    srand(time(0));
    int b = rand() % 10;
    return ((a + b) % 3 == 0) && (a % 2 == 0);
}
```

```
void kontrol_et(int a) {
    if (karmaşık_boolean(a)) {
        std::cout << "Koşul sağlandı, karmaşık boolean: true" << std::endl;
    } else {
        std::cout << "Koşul sağlanmadı, karmaşık boolean: false" << std::endl;
    }
}
```

```
int main() {
    kontrol_et(12); // Girdi değeriyle fonksiyon çağrılıyor
    kontrol_et(7);  // Farklı girdi değeriyle tekrar çağrılıyor
    return 0;
}
```

Özet:

Bu örnekte, boolean değeri hesaplamak için kullanılan karmaşık bir koşul seti bulunur. Bu şartlar, kodun anlaşılmasını ve analiz edilmesini zorlaştırır, çünkü her defasında farklı sonuçlar üretilebilir ve koşulların ne zaman sağlandığını anlamak güçleşir.

1.8. Fonksiyon Boolean Return Kodlarını Karmaşıklaştırma (Function Boolean Return Codes)

Teorik Açıklama:

Fonksiyonların dönüş değerlerini karmaşıklaştırmak, kodun akışını ve işlevselliğini anlamayı zorlaştırır. Bu teknik, fonksiyonların hangi koşullar altında hangi değerleri döndürdüğünü belirsiz hale getirir.

Örnek Önerisi:

- Karmaşık matematiksel işlemlerle koşullu dönüş değerleri üreten bir fonksiyon.
- Geri mühendislik işlemine karşı fonksiyonların dönüş değerlerini tahmin edilemez hale getirme.

1.8.1. Karmaşık Matematiksel İşlemlerle Koşullu Dönüş Değerleri Üreten Bir Fonksiyon

Açıklama:

Bu örnekte, fonksiyonun dönüş değeri karmaşık matematiksel işlemlerle belirlenir. Bu, fonksiyonun ne zaman `true` veya `false` döndürdüğünü anlamayı zorlaştırarak kodun analiz edilmesini güçleştirir.

```cpp
#include <iostream>
#include <cmath> // Matematiksel fonksiyonlar için

// Karmaşık matematiksel işlemle boolean döndüren fonksiyon
bool karmaşık_donus_degeri(int a, int b) {
    int sonuc = static_cast<int>(pow(a, 3) - pow(b, 2)); // Karmaşık hesaplama
    return (sonuc % 7 == 0); // Koşula göre boolean döner
}
```

```
void kontrol_et(int a, int b) {
    if (karmaşık_donus_degeri(a, b)) {
        std::cout << "Karmaşık dönüş değeri: true" << std::endl;
    } else {
        std::cout << "Karmaşık dönüş değeri: false" << std::endl;
    }
}
```

```
int main() {
    kontrol_et(4, 2);  // Girdi değerleriyle fonksiyon çağrılıyor
    kontrol_et(7, 3);  // Farklı girdi değerleriyle tekrar çağrılıyor
    return 0;
}
```

Özet:

Bu kodda, `karmaşık_donus_degeri` fonksiyonu `a` ve `b` değerlerine göre karmaşık matematiksel işlemler gerçekleştirir. Hesaplamalar sonucunda `sonuc % 7 == 0` koşuluna göre `true` veya `false` döner. Bu, fonksiyonun ne zaman hangi değeri döndürdüğünü anlamayı zorlaştırır ve geri mühendislik işlemlerini karmaşık hale getirir.

1.8.2. Geri Mühendislik İşlemine Karşı Fonksiyonların Dönüş Değerlerini Tahmin Edilemez Hale Getirme

Açıklama:

Bu örnekte, fonksiyonların dönüş değerleri rastgele işlemlerle belirsiz hale getirilir.

Böylece, fonksiyonların ne zaman hangi değeri döndürdüğü dışarıdan anlaşılmaz.

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

// Geri mühendisliği zorlaştırmak için rastgele işlemlerle boolean döndüren fonksiyon
bool tahmin_edilemez_donus(int a, int b) {
    srand(time(0));
    int rastgele = rand() % 100;
    int sonuc = (a * b) + rastgele; // Rastgele sayı ile işlem
    return (sonuc % 5 == 0); // Koşula göre true veya false döner
}
```

```
void sonuc_kontrol(int a, int b) {
    if (tahmin_edilemez_donus(a, b)) {
        std::cout << "Tahmin edilemez dönüş değeri: true" << std::endl;
    } else {
        std::cout << "Tahmin edilemez dönüş değeri: false" << std::endl;
    }
}
```

```
int main() {
    sonuc_kontrol(6, 4); // Farklı girdi değerleriyle fonksiyon çağrılıyor
    sonuc_kontrol(7, 5);
    return 0;
}
```

Özet:

Bu örnekte, `tahmin_edilemez_donus` fonksiyonu rastgele bir sayı üreterek hesaplamalarına dahil eder. Bu rastgelelik, fonksiyonun ne zaman `true` veya `false` döndüreceğini tahmin etmeyi imkansız hale getirir. Bu yaklaşım, geri mühendislik ve statik analiz araçlarına karşı fonksiyonları daha korunaklı hale getirir.

1.9. Fonksiyon Parametrelerinin Gizlenmesi (Obfuscation of Function Parameters)

Teorik Açıklama:

Fonksiyon parametrelerini gizlemek, fonksiyonların aldığı verilerin ne olduğunu ve nasıl işlendiğini anlamayı zorlaştırır. Bu teknik, parametrelerin anlamını ve kullanımını belirsiz hale getirir.

Örnek Önerisi:

- Parametre isimlerinin anlamsız hale getirildiği ve fonksiyonların işlevinin dışarıdan anlaşılmaz olduğu bir örnek.
- Parametrelerin maskeleme yöntemleriyle gizlenmesi.

1.9.1. Parametre İsimlerinin Anlamsız Hale Getirilmesi

Açıklama:

Bu teknik, fonksiyon parametrelerinin isimlerini anlamsız hale getirerek kodun anlaşılmasını zorlaştırır. Dışarıdan bakıldığında, fonksiyonun ne yaptığı veya parametrelerin ne amaçla kullanıldığı anlaşılmaz hale gelir.

```cpp
#include <iostream>

// Anlamsız parametre isimleri ile tanımlanan fonksiyon
int z4m1nq0(int p1, int p2) {
    return p1 * p2 + (p1 - p2); // Karmaşık bir işlem
}
```

```cpp
int main() {
    int sonuc = z4m1nq0(10, 5); // Fonksiyon çağrısı
    std::cout << "Sonuç: " << sonuc << std::endl;
    return 0;
}
```

Özet:

Bu örnekte, `z4m1nq0` gibi anlamsız bir fonksiyon ismi ve `p1`, `p2` gibi parametre isimleri kullanılmıştır. Parametrelerin ne olduğu ve nasıl kullanıldığı anlaşılmaz hale getirilmiştir. Bu, kodun geri mühendislik ile analiz edilmesini zorlaştırır.

1.9.2. Parametrelerin Maskeleme Yöntemleri ile Gizlenmesi

Açıklama:

Bu teknikte, parametreler maskeleme işlemi ile gizlenir. Parametreler çalışma zamanında açığa çıkarılır ve kodun ne yaptığı dışarıdan bakıldığında anlaşılamaz.

```
#include <iostream>

// Maskeleme fonksiyonu
int parametre_maskele(int param) {
    return param ^ 0x5A; // XOR ile basit bir maskeleme
}
```

```
// Gizlenmiş parametre ile işlem yapan fonksiyon
int gizli_fonksiyon(int a, int b) {
    int gercek_a = parametre_maskele(a); // Parametre maskelemesini çöz
    int gercek_b = parametre_maskele(b);
    return gercek_a + gercek_b; // Gerçek değerlerle işlem yap
}
```

```
int main() {
    int a = parametre_maskele(10); // Parametreler maskelenmiş
    int b = parametre_maskele(20);
    int sonuc = gizli_fonksiyon(a, b); // Gizli fonksiyon çağrısı
    std::cout << "Sonuç: " << sonuc << std::endl;
    return 0;
}
```

Özet:

Bu örnekte, `parametre_maskele` fonksiyonu parametreleri maskeler ve daha sonra maskeleme çözülerek fonksiyon içinde gerçek değerler kullanılır. Bu teknik, fonksiyonun aldığı parametrelerin ne olduğunu gizler ve saldırganların analiz etmesini zorlaştırır.

1.10. Anlamsız Parametreler ve İşlemler Ekleyerek Kodun Analizini Zorlaştırma (Bogus Function Parameters & Operations)

Teorik Açıklama:

Kodun analizini zorlaştırmak için fonksiyonlara anlamsız parametreler ve gereksiz işlemler eklemek kullanılır. Bu teknik, saldırganların fonksiyonların gerçek amacını belirlemesini engeller.

Örnek Önerisi:

- Fonksiyonlara gereksiz parametreler ekleyerek ve anlamsız işlemler yaparak kodun karmaşık hale getirildiği bir örnek.
- Saldırganları yanıltacak anlamsız hesaplama ve koşulların eklendiği bir fonksiyon.

1.10.1. Fonksiyonlara Gereksiz Parametreler Ekleyerek Kodun Karmaşık Hale Getirilmesi

Açıklama:

Bu teknik, fonksiyonlara gereksiz parametreler ekleyerek kodun anlaşılmasını zorlaştırır.

Parametrelerin işlevi olmadığından, fonksiyonun gerçek amacı belirsiz hale gelir.

```cpp
#include <iostream>

// Gereksiz parametreler içeren fonksiyon
int anlamsiz_fonksiyon(int a, int b, int gereksiz1, int gereksiz2) {
    // Gerçek işlem sadece a ve b ile yapılır
    return (a * b) + 10;
}
```

```
int main() {
    // Gereksiz parametrelerle fonksiyon çağrısı
    int sonuc = anlamsiz_fonksiyon(5, 3, 100, 200);
    std::cout << "Sonuç: " << sonuc << std::endl;
    return 0;
}
```

Özet:

Bu kodda, `anlamsiz_fonksiyon` adlı fonksiyona gereksiz parametreler (`gereksiz1` , `gereksiz2`) eklenmiştir. Bu parametrelerin gerçek bir işlevi olmadığı için, dışarıdan bakıldığında fonksiyonun ne yaptığı anlaşılmaz. Bu teknik, geri mühendislik işlemlerini zorlaştırır.

1.10.2. Anlamsız Hesaplama ve Koşulların Eklendiği Bir Fonksiyon

Açıklama:

Bu örnekte, fonksiyona anlamsız işlemler ve gereksiz koşullar eklenerek kod karmaşık hale getirilir. Bu yaklaşım, fonksiyonun gerçek amacını gizleyerek analiz edilmesini zorlaştırır.

```cpp
#include <iostream>

// Anlamsız işlemler ve koşullar içeren fonksiyon
int karmasik_fonksiyon(int a, int b, int c) {
    int temp = a * b; // Gerçek işlem
    if (c > 100) { // Anlamsız koşul
        temp += c; // Anlamsız işlem
    }
    for (int i = 0; i < c; i++) { // Gereksiz döngü
        temp -= i; // Anlamsız hesaplama
    }
    return temp;
}
```

```
int main() {
    int sonuc = karmasik_fonksiyon(5, 3, 50); // Fonksiyon çağrısı
    std::cout << "Sonuç: " << sonuc << std::endl;
    return 0;
}
```

Özet:

Bu kodda, fonksiyonun ana işlemi `a * b` ile yapılır. Ancak gereksiz koşullar (`if (c > 100)`) ve döngüler eklenerek kod karmaşık hale getirilmiştir. Anlamsız işlemler ve hesaplamalar, saldırganların fonksiyonun gerçek işlevini anlamasını zorlaştırır.

1.11. Kontrol Akışını Düzleştirerek Tahmin Edilemez Hale Getirme (Control Flow Flattening)

Teorik Açıklama:

Kontrol akışını düzleştirmek, kodun normal akışını bozar ve programın akışını tahmin etmeyi zorlaştırır. Bu teknik, kontrol yapılarını karmaşıklaştırarak kodun analizini zorlaştırır.

Örnek Önerisi:

- Düzleştirilmiş kontrol akışıyla koşullu ifadeler yerine durum tabanlı geçişlerin kullanıldığı bir örnek.
- Kod akışının tahmin edilemez hale getirilmesi.

1.11.1. Durum Tabanlı Geçişlerin Kullanıldığı Düzleştirilmiş Kontrol Akışı

Açıklama:

Bu örnekte, kontrol akışı düzleştirilmiş ve durum tabanlı geçişler kullanılarak kodun akışı karmaşık hale getirilmiştir. Bu yöntem, koşullu ifadeler yerine durumlar üzerinden ilerler ve kodun normal akışı bozulur.

```cpp
#include <iostream>

void kontrol_akisi_duzlestir(int a) {
    int state = 0; // Başlangıç durumu
    while (true) {
        switch (state) {
            case 0:
                if (a > 10) {
                    state = 1; // Durum 1'e geç
                } else {
                    state = 2; // Durum 2'ye geç
                }
                break;
            case 1:
                std::cout << "Durum 1: a > 10" << std::endl;
                state = 3; // Son duruma geç
                break;
            case 2:
                std::cout << "Durum 2: a <= 10" << std::endl;
                state = 3; // Son duruma geç
                break;
            case 3:
                return; // Program sona erer
        }
    }
}
```

```
int main() {
    kontrol_akisi_duzlestir(12); // Girdi değerine göre durum tabanlı akış
    kontrol_akisi_duzlestir(8);  // Farklı girdi ile çağrılıyor
    return 0;
}
```

Özet:

Bu örnekte, kontrol akışı düzleştirilmiştir ve `state` değişkeni ile duruma bağlı olarak hareket edilir. Koşullu ifadeler yerine, durum geçişleri yapılır. Bu, programın akışını tahmin etmeyi zorlaştırır, çünkü durum tabanlı bir yapı kullanıldığında hangi durumda hangi işlem yapılacağı dışarıdan görünmez hale gelir.

1.11.2. Kod Akışını Tahmin Edilemez Hale Getirme

Açıklama:

Bu örnekte, kontrol akışını düzleştirmek için tahmin edilemez durumlar eklenmiştir. Bu durum, kodun analizini zorlaştırır ve saldırganların kod akışını anlamasını engeller.

```cpp
#include <iostream>
#include <cstdlib>
#include <ctime>

// Rastgele durumlar ile kontrol akışı
void tahmin_edilemez_kontrol_akisi(int a) {
    srand(time(0)); // Rastgele sayı üreteci
    int state = rand() % 3; // Rastgele başlangıç durumu

    while (true) {
        switch (state) {
            case 0:
                std::cout << "Başlangıç durumu, a: " << a << std::endl;
                if (a > 5) {
                    state = 1; // Durum 1'e geç
                } else {
                    state = 2; // Durum 2'ye geç
                }
                break;
            case 1:
                std::cout << "Durum 1: a > 5" << std::endl;
                state = rand() % 3; // Rastgele durum değiştir
                break;
            case 2:
                std::cout << "Durum 2: a <= 5" << std::endl;
                state = rand() % 3; // Rastgele durum değiştir
                break;
            case 3:
                std::cout << "Program sona eriyor." << std::endl;
                return; // Program biter
        }
    }
}
```

```
int main() {
    tahmin_edilemez_kontrol_akisi(7); // Fonksiyon çağrısı
    return 0;
}
```

Özet:

Bu kodda, rastgele durumlarla kontrol akışı yönetilir. Program her çalıştığında farklı bir başlangıç durumu ve farklı bir akış oluşabilir. Bu durum, kodun akışını anlamayı ve analiz etmeyi zorlaştırır. Kod, tahmin edilemez hale gelerek saldırganlar için daha güvenli olur.

1.12. Çıkış Noktalarını Rastgele Hale Getirerek Kodun Öngörülebilirliğini Azaltma (Randomized Exit Points)

Teorik Açıklama:

Çıkış noktalarını rastgele hale getirmek, kodun ne zaman sona ereceğini belirsizleştirir. Bu teknik, programın kontrol akışını tahmin etmeyi zorlaştırır ve analiz araçlarının işini karmaşıklaştırır.

Örnek Önerisi:

- Rastgele belirlenen çıkış noktalarıyla programın beklenmedik yerlerde sona erdiği bir örnek.
- Programın farklı koşullarda farklı çıkış noktalarına sahip olması.

Sona Ermesi

Açıklama:

Bu örnekte, programın rastgele belirlenen koşullara bağlı olarak farklı çıkış noktalarına sahip olduğu bir yapı oluşturulmuştur. Bu yaklaşım, programın ne zaman ve nerede sona ereceğini belirsiz hale getirir.

```
#include <iostream>
#include <cstdlib>
#include <ctime>

// Rastgele çıkış noktası belirleyen fonksiyon
void rastgele_cikis(int a) {
    srand(time(0)); // Rastgele sayı üreteci başlatılıyor
    int random_exit = rand() % 3; // 0, 1 veya 2 arasında rastgele değer

    std::cout << "İşlem başlatıldı..." << std::endl;

    if (a > 10 && random_exit == 0) {
        std::cout << "Çıkış Noktası 1" << std::endl;
        return; // Program burada sona erer
    }

    if (a < 5 && random_exit == 1) {
        std::cout << "Çıkış Noktası 2" << std::endl;
        return; // Program burada sona erer
    }

    std::cout << "Normal işleyiş devam ediyor." << std::endl;
```

```
int main() {
    rastgele_cikis(12); // Farklı girişler ile test ediliyor
    rastgele_cikis(3);
    rastgele_cikis(7);
    return 0;
}
```

Özet:

Bu kodda, `rastgele_cikis` fonksiyonu rastgele bir çıkış noktası seçer. Eğer belirlenen rastgele koşullar sağlanırsa, program beklenmedik bir noktada sona erer. Bu yaklaşım, kodun öngörülebilirliğini azaltır ve analiz araçları için programın akışını takip etmeyi zorlaştırır.

Açıklama:

Bu örnekte, programın farklı koşullarda rastgele çıkış noktalarına sahip olması sağlanmıştır. Bu, kodun akışını tahmin etmeyi zorlaştırır ve kodun analizi sırasında programın tam olarak ne zaman sona ereceğini belirsiz hale getirir.

```
#include <iostream>
#include <cstdlib>
#include <ctime>

// Farklı koşullara göre çıkış yapan fonksiyon
void tahmin_edilemez_cikis(int a, int b) {
    srand(time(0)); // Rastgele sayı üretici
    int random_exit = rand() % 4; // 0, 1, 2 veya 3

    if (a > b && random_exit == 0) {
        std::cout << "Çıkış Noktası 1: a > b" << std::endl;
        return; // Program sona erer
    }

    if (b > a && random_exit == 1) {
        std::cout << "Çıkış Noktası 2: b > a" << std::endl;
        return; // Program burada sona erer
    }

    if (a == b && random_exit == 2) {
        std::cout << "Çıkış Noktası 3: a == b" << std::endl;
        return; // Program burada sona erer
    }

    std::cout << "Program normal şekilde sona erdi." << std::endl;
```

```c
int main() {
    tahmin_edilemez_cikis(5, 10); // Farklı girişler ile test ediliyor
    tahmin_edilemez_cikis(10, 5);
    tahmin_edilemez_cikis(7, 7);
    return 0;
}
```

Özet:

Bu kodda, program farklı giriş değerlerine ve rastgele seçilen çıkış noktalarına göre sona erer. `tahmin_edilemez_cikis` fonksiyonu, hem giriş koşullarına hem de rastgele sayılara göre çıkış yapar. Bu yapı, programın ne zaman sona ereceğini tahmin etmeyi zorlaştırarak kodun analiz edilmesini engeller.

1.13. Son Sürümde Loglamaların Devre Dışı Bırakılması (Logging Disabled on Release)

Teorik Açıklama:

Loglama, geliştirme sürecinde faydalı olsa da, son sürümlerde devre dışı bırakılması güvenlik açısından önemlidir. Loglar, hassas bilgileri açığa çıkarabilir ve saldırganların sistem hakkında bilgi edinmesini kolaylaştırabilir. Bu nedenle, loglamanın yalnızca geliştirme aşamasında aktif olması sağlanmalı ve son sürümlerde kapatılmalıdır.

Örnek Önerisi:

- Derleme aşamasında `DEBUG` veya `RELEASE` modlarına göre loglamayı devre dışı bırakan bir makro örneği.
- Son sürümde loglamaların tamamen kaldırıldığı bir uygulama.

1.13.1. Derleme Aşamasında `DEBUG` veya `RELEASE` Modlarına Göre Loglamayı Devre Dışı Bırakan Bir Makro Örneği

Açıklama:

Bu örnekte, loglama işlemleri `DEBUG` veya `RELEASE` modlarına bağlı olarak kontrol edilir. Geliştirme sırasında (`DEBUG`) loglamalar aktif, son sürümde (`RELEASE`) devre dışı bırakılır.

```
#include <iostream>

// DEBUG veya RELEASE modlarına göre loglama kontrolü
#ifdef DEBUG
    #define LOG(x) std::cout << "LOG: " << x << std::endl;
#else
    #define LOG(x) // Boş tanım, loglama yapılmaz
#endif
```

```
void sistem_bilgisi() {
    LOG("Sistem bilgileri alınıyor...");
    // Diğer işlemler...
    std::cout << "Sistem işlemleri tamamlandı." << std::endl;
}
```

```
int main() {
    sistem_bilgisi();
    return 0;
}
```

Özet:

Bu kodda, `DEBUG` modunda loglamalar aktifken, `RELEASE` modunda loglama makrosu boş tanımlanarak loglamalar devre dışı bırakılır. Bu, son sürümde logların çıkmasını önler ve hassas bilgilerin ifşa olmasını engeller.

1.13.2. Son Sürümde Loglamaların Tamamen Kaldırıldığı Bir Uygulama

Açıklama:

Bu örnek, son sürümde tüm loglama işlemlerinin tamamen kaldırıldığı bir yapı içerir. Geliştirme aşamasında aktif olan loglar, son sürüme geçildiğinde derleme sırasında tamamen devre dışı bırakılır.

```cpp
#include <iostream>

// DEBUG modunda loglama aktif, RELEASE modunda devre dışı
void kritik_islem() {
#ifdef DEBUG
    std::cout << "DEBUG: Kritik işlem başlatıldı." << std::endl;
#endif
    // Kritik işlemler burada gerçekleştirilir
    std::cout << "Kritik işlem tamamlandı." << std::endl;
}
```

```
int main() {
    kritik_islem();
    return 0;
}
```

Özet:

Bu kodda, kritik işlemler sırasında yalnızca `DEBUG` modunda loglamalar görünür. `RELEASE` modunda loglama kodu derleme aşamasında tamamen kaldırılır, bu sayede son sürümde loglama işlemi gerçekleşmez ve hassas verilerin sızdırılması önlenir.

2. Java ve Yorumlanan Diller İçin Kod Güçlendirme Teknikleri

Java ve diğer yorumlanan dillerde kod güçlendirme, güvenlik açıklarını azaltmak ve geri mühendislik işlemlerini zorlaştırmak için kullanılır.

2.1. Proguard ile Kod Obfuske ve Koruma (Proguard Code Obfuscation and Code Shrink Protection)

Teorik Açıklama: Proguard, Java kodlarını küçültme, optimize etme ve obfuske ederek kodun analiz edilmesini zorlaştırır.

Uygulama Örnekleri:

1. Proguard yapılandırma dosyası ile kodun küçültülmesi ve optimize edilmesi.
2. Obfuske edilmiş kodun test edilmesi ve hataların çözülmesi.
3. Proguard raporlarının analizi ile hangi öğelerin obfuske edildiğinin tespiti.

2.1.1 Proguard yapılandırma dosyası ile kodun küçültülmesi ve optimize edilmesi

1. Mobil Android Projesi (Gradle)

Dosya Yapısı:

```
/MyAndroidApp
├── app
│   ├── src
│   │   ├── main
│   │   │   ├── java/com/example/myandroidapp/MainActivity.java
│   │   │   └── res/layout/activity_main.xml
│   ├── build.gradle
│   ├── proguard-rules.pro
├── build.gradle
├── settings.gradle
```

Proje Dosyaları:

app/build.gradle

```
android {
    compileSdkVersion 30
    defaultConfig {
        applicationId "com.example.myandroidapp"
        minSdkVersion 21
        targetSdkVersion 30
        versionCode 1
        versionName "1.0"
    }

    buildTypes {
        release {
            minifyEnabled true
            shrinkResources true
            proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'
        }
    }
}
```

proguard-rules.pro

```
# Keep MainActivity class and its methods
-keep class com.example.myandroidapp.MainActivity { *; }
```

MainActivity.java

```
package com.example.myandroidapp;

import android.os.Bundle;
import androidx.appcompat.app.AppCompatActivity;

public class MainActivity extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);
    }
}
```

activity_main.xml

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout
    xmlns:android="http://schemas.android.com/apk/res/android"
    android:layout_width="match_parent"
    android:layout_height="match_parent">

    <TextView
        android:id="@+id/textView"
        android:layout_width="wrap_content"
        android:layout_height="wrap_content"
        android:text="Hello, Proguard!"
        android:textSize="20sp"/>
</androidx.constraintlayout.widget.ConstraintLayout>
```

Nasıl Çalıştırılır:

- Android Studio ile projeyi açın.
- `Build > Generate Signed Bundle / APK` ile `release` APK oluşturun.
- Proguard otomatik olarak kodu obfuske eder.

2. Masaüstü Java Projesi (Gradle)

Dosya Yapısı:

```
/MyDesktopApp
├── src
│   └── main
│       └── java/com/example/mydesktopapp/Main.java
├── build.gradle
├── proguard-rules.pro
```

Proje Dosyaları:

build.gradle

```
plugins {
    id 'application'
}

application {
    mainClass = 'com.example.mydesktopapp.Main'
}

task proguard(type: JavaExec) {
    main = 'proguard.ProGuard'
    classpath = configurations.proguard
    args = '-injars', "$buildDir/classes/java/main", '-outjars', "$buildDir/classes/obfuscated.jar", '-libraryjars', "$JAVA_HOME/jre/lib/rt.jar", '@proguard-rules.pro'
}

configurations {
    proguard
}

dependencies {
    proguard 'net.sf.proguard:proguard-base:6.0.3'
}
```

proguard-rules.pro

```
-keep class com.example.mydesktopapp.Main { *; }
```

Main.java

```
package com.example.mydesktopapp;

public class Main {
    public static void main(String[] args) {
        System.out.println("Hello, Proguard!");
    }
}
```

Nasıl Çalıştırılır:

- Terminalde projenin bulunduğu dizinde şu komutu çalıştırın

```
gradlew proguard
```

Obfuske edilmiş kodu `build/classes/obfuscated.jar` içinde bulabilirsiniz.

3. JavaCard Projesi (Maven)

Dosya Yapısı:

```
/MyJavaCardApp
├── src
│   └── main
│       └── java/com/example/myjavacardapp/MyJavaCardApplet.java
├── pom.xml
├── proguard-javacard-rules.pro
```

Proje Dosyaları:

pom.xml

```
<project xmlns="http://maven.apache.org/POM/4.0.0"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
        http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.example</groupId>
    <artifactId>myjavacardapp</artifactId>
    <version>1.0-SNAPSHOT</version>

    <build>
        <plugins>
            <plugin>
                <groupId>com.github.wvengen</groupId>
                <artifactId>proguard-maven-plugin</artifactId>
                <version>2.0.15</version>
                <executions>
                    <execution>
                        <goals>
                            <goal>proguard</goal>
                        </goals>
                    </execution>
                </executions>
            </plugin>
        </plugins>
    </build>
</project>
```

proguard-javacard-rules.pro

```
-keep class javacard.framework.Applet { *; }
-keep class com.example.myjavacardapp.MyJavaCardApplet { *; }
```

MyJavaCardApplet.java

```
package com.example.myjavacardapp;

import javacard.framework.*;

public class MyJavaCardApplet extends Applet {
    public static void install(byte[] bArray, short bOffset, byte bLength) {
        new MyJavaCardApplet().register(bArray, (short) (bOffset + 1), bArray[bOffset]);
    }
}
```

Nasıl Çalıştırılır:

- Terminalde projenin bulunduğu dizinde şu komutu çalıştırın

```
mvn clean package
```

Maven, Proguard ile kodu küçülterek `target` dizininde obfuske edilmiş bir jar dosyası oluşturur.

2. Masaüstü Java Projesi (Gradle)

Proguard Yapılandırma Dosyası ile Kodun Küçültülmesi ve Optimize Edilmesi

Açıklama:

Proguard, masaüstü Java uygulamalarında kullanılmayan kodları kaldırarak, kodu küçültmek ve optimize etmek için kullanılır. Bu işlem, uygulamanın boyutunu azaltır ve sınıfları, metotları, ve değişkenleri anlamsız isimlerle değiştirerek kodun analiz edilmesini zorlaştırır.

Yapılandırma dosyası: `proguard-rules.pro`

```
-keep class com.example.mydesktopapp.Main { *; }
-keepclassmembers class * {
    public <init>(...);
}
```

- **Açıklama:**
  Bu yapılandırmada `Main` sınıfı korunuyor ve obfuske edilmiyor. Proguard'ın diğer sınıfları küçültmesine ve optimize etmesine izin veriliyor. Kurallar, belirli sınıfları korurken geri kalanları obfuske etmeyi sağlar.

Obfuske Edilmiş Kodun Test Edilmesi ve Hataların Çözülmesi

1. **Kodun Derlenmesi ve Obfuske Edilmesi:**
    - Terminalde projenizin bulunduğu dizinde şu komutu çalıştırarak Proguard ile derleme işlemini başlatın:

```
gradlew proguard
```

2. **Test Adımları:**

   - `build/classes/obfuscated.jar` dosyasına göz atın ve programın çalışıp çalışmadığını doğrulamak için test edin.
   - Eğer uygulama beklendiği gibi çalışmıyorsa ve bazı kritik sınıflar veya metotlar obfuske edilmişse, `proguard-rules.pro` dosyasına o sınıf ve metotları koruyacak ek kurallar ekleyin.
   - Örnek: Eğer `myMethod()` metodu çalışmıyorsa şu kuralı ekleyebilirsiniz:

```
-keep class com.example.mydesktopapp.MyClass {
    public void myMethod();
}
```

Proguard Raporlarının Analizi ile Hangi Öğelerin Obfuske Edildiğinin Tespiti

1. **Mapping Dosyası:** Proguard, `mapping.txt` adında bir rapor oluşturur. Bu dosya, hangi sınıfların, metodların ve değişkenlerin obfuske edildiğini ve hangi isimlerle değiştirildiğini gösterir. `mapping.txt` dosyasını analiz ederek, hangi öğelerin korunduğunu ve hangilerinin obfuske edildiğini öğrenebilirsiniz.

Örnek Mapping Dosyası:

```
com.example.mydesktopapp.Main -> a:
    void main(java.lang.String[]) -> a
    int myVariable -> b
```

- **Açıklama:**
  Bu örnekte `com.example.mydesktopapp.Main` sınıfı `a` olarak, `myVariable` ise `b` olarak değiştirilmiştir. Mapping dosyası, hata ayıklamak veya korunan sınıfların durumunu kontrol etmek için kullanılır.

3. JavaCard Projesi (Maven)

Proguard Yapılandırma Dosyası ile Kodun Küçültülmesi ve Optimize Edilmesi

Açıklama:

JavaCard projelerinde, Proguard kullanılarak sınıflar ve metotlar küçültülür ve optimize edilir. JavaCard'ın sınırlı kaynakları nedeniyle kod küçültme ve optimize işlemleri büyük önem taşır.

Yapılandırma dosyası: `proguard-javacard-rules.pro`

```
-keep class javacard.framework.Applet { *; }
-keep class com.example.myjavacardapp.MyJavaCardApplet { *; }
```

- **Açıklama:**
  Bu yapılandırmada JavaCard framework'ü içinde yer alan `Applet` sınıfı ve uygulamamızdaki `MyJavaCardApplet` sınıfı korunuyor. Geri kalan sınıflar küçültülecek ve obfuske edilecektir.

Obfuske Edilmiş Kodun Test Edilmesi ve Hataların Çözülmesi

1. **Kodun Derlenmesi ve Obfuske Edilmesi:**
    - Terminalde şu komutu çalıştırarak Maven ile Proguard işlemini başlatın:

```
mvn clean package
```

2. **Test Adımları:**
   - `target/myjavacardapp-obfuscated.jar` dosyasına göz atın ve uygulamanın doğru çalıştığını test edin.
   - Eğer bazı sınıflar ya da metotlar gerektiği şekilde çalışmıyorsa, bu sınıfları `proguard-javacard-rules.pro` dosyasına ekleyerek koruyabilirsiniz.
   - Örnek: Eğer `javacard.framework.ISO7816` sınıfı obfuske edilmişse ve hataya neden oluyorsa, bu sınıfı koruma kuralı ekleyin:

```
-keep class javacard.framework.ISO7816 { *; }
```

Proguard Raporlarının Analizi ile Hangi Öğelerin Obfuske Edildiğinin Tespiti

1. **Mapping Dosyası:** Maven ile oluşturulan Proguard raporu, `mapping.txt` dosyası altında bulunur. Bu dosya, hangi sınıfların, metodların ve değişkenlerin isimlerinin obfuske edildiğini gösterir.

Örnek Mapping Dosyası:

```
com.example.myjavacardapp.MyJavaCardApplet -> a:
    void install(byte[], short, byte) -> a
```

- **Açıklama:**
  `com.example.myjavacardapp.MyJavaCardApplet` sınıfı `a` olarak değiştirilmiş, `install()` metodu ise `a` olarak adlandırılmıştır. Bu rapor sayesinde kodun nasıl obfuske edildiğini görebilir ve hangi öğelerin değiştirilip değiştirilmediğini kontrol edebilirsiniz.

2.2. Cihaz Bağlama İçin Ayrı Parmak İzi Depolama (Separated Fingerprint Storage for Device Binding)

Teorik Açıklama: Cihazın benzersiz özelliklerini kullanarak, uygulamanın yalnızca belirli bir cihazda çalışmasını sağlamak için kullanılan bir tekniktir.

Uygulama Örnekleri:

1. Cihaz parmak izinin şifrelenerek güvenli bir şekilde depolanması.
2. Parmak izi doğrulaması ile uygulamanın cihaz üzerinde çalışmasını sağlama.
3. Parmak izi verilerinin gizlenmesi ve saldırılara karşı korunması.

2.2.1. Cihaz Parmak İzinin Şifrelenerek Güvenli Bir Şekilde Depolanması

Açıklama:

Cihazın benzersiz bir parmak izi, genellikle donanım veya sistem özelliklerinden alınan bilgilerden oluşturulur. Bu parmak izi şifrelenerek güvenli bir şekilde cihazda saklanır.

Java Örneği:

```
import java.nio.charset.StandardCharsets;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.util.Base64;

public class DeviceFingerprint {

    // Benzersiz bir cihaz parmak izi oluşturma
    public static String generateFingerprint(String deviceID, String hardwareSerial) throws NoSuchAlgorithmException {
        String rawFingerprint = deviceID + hardwareSerial;

        // SHA-256 ile cihaz parmak izini şifreleme
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] encodedHash = digest.digest(rawFingerprint.getBytes(StandardCharsets.UTF_8));

        // Şifrelenmiş parmak izini Base64 ile encode etme
        return Base64.getEncoder().encodeToString(encodedHash);
    }
```

```
    public static void main(String[] args) {
        try {
            // Benzersiz cihaz bilgileri
            String deviceID = "device12345";
            String hardwareSerial = "hwserial67890";

            // Parmak izi oluşturulup şifreleniyor
            String fingerprint = generateFingerprint(deviceID, hardwareSerial);
            System.out.println("Şifrelenmiş Parmak İzi: " + fingerprint);

        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }
    }
}
```

Özet:

- `generateFingerprint` metodu, cihazdan elde edilen `deviceID` ve `hardwareSerial` bilgilerini kullanarak bir cihaz parmak izi oluşturur.
- Parmak izi SHA-256 algoritmasıyla şifrelenir ve Base64 formatında saklanabilir

2.2.2. Parmak İzi Doğrulaması ile Uygulamanın Cihaz Üzerinde Çalışmasını Sağlama

Açıklama:

Cihaz parmak izi, yalnızca belirli bir cihazda uygulamanın çalışmasını sağlamak için doğrulanır. Parmak izi eşleşmediğinde uygulama çalıştırılmaz.

Java Örneği:

```
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.nio.charset.StandardCharsets;
import java.util.Base64;

public class DeviceFingerprintCheck {

    // Daha önce saklanan şifrelenmiş parmak izi
    private static final String STORED_FINGERPRINT = "storedFingerprintValue"; // Bu değeri parmak izi şifreleme sırasında alıyoruz
```

```
    public static String generateFingerprint(String deviceID, String hardwareSerial) throws NoSuchAlgorithmException {
        String rawFingerprint = deviceID + hardwareSerial;

        // Parmak izini tekrar oluşturma
        MessageDigest digest = MessageDigest.getInstance("SHA-256");
        byte[] encodedHash = digest.digest(rawFingerprint.getBytes(StandardCharsets.UTF_8));

        return Base64.getEncoder().encodeToString(encodedHash);
    }
```

```
    public static boolean verifyFingerprint(String deviceID, String hardwareSerial) throws NoSuchAlgorithmException {
        // Şu anki cihazın parmak izi
        String currentFingerprint = generateFingerprint(deviceID, hardwareSerial);

        // Daha önce saklanan parmak izi ile karşılaştırma
        return STORED_FINGERPRINT.equals(currentFingerprint);
    }
```

```
    public static void main(String[] args) {
        try {
            // Gerçek cihaz bilgileri (örnek olarak)
            String deviceID = "device12345";
            String hardwareSerial = "hwserial67890";

            // Parmak izi doğrulama
            if (verifyFingerprint(deviceID, hardwareSerial)) {
                System.out.println("Parmak izi doğrulandı, uygulama bu cihazda çalışabilir.");
            } else {
                System.out.println("Parmak izi doğrulanamadı, uygulama bu cihazda çalışamaz.");
            }

        } catch (NoSuchAlgorithmException e) {
            e.printStackTrace();
        }
    }
}
```

Özet:

- `verifyFingerprint` metodu, mevcut cihazın parmak izini daha önce depolanan parmak izi ile karşılaştırır.
- Eğer parmak izleri eşleşirse, uygulama cihazda çalışabilir.

2.2.3. Parmak İzi Verilerinin Gizlenmesi ve Saldırılara Karşı Korunması

Açıklama:

Parmak izi verileri güvenli bir şekilde saklanmalıdır. Şifrelenmiş parmak izi verisi bir dosyada veya güvenli bir veri deposunda saklanabilir.

Java Örneği:

```
import javax.crypto.Cipher;
import javax.crypto.KeyGenerator;
import javax.crypto.SecretKey;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;

public class SecureFingerprintStorage {

    // Şifreleme anahtarı (Güvenli bir şekilde saklanmalı)
    private static final String KEY = "1234567890123456"; // 16-byte key
```

```
// Cihaz parmak izini AES ile şifreleme
public static String encryptFingerprint(String fingerprint) throws Exception {
    SecretKey secretKey = new SecretKeySpec(KEY.getBytes(), "AES");
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.ENCRYPT_MODE, secretKey);
    byte[] encryptedData = cipher.doFinal(fingerprint.getBytes());
    return Base64.getEncoder().encodeToString(encryptedData);
}
```

```
// Şifrelenmiş cihaz parmak izini çözme
public static String decryptFingerprint(String encryptedFingerprint) throws Exception {
    SecretKey secretKey = new SecretKeySpec(KEY.getBytes(), "AES");
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.DECRYPT_MODE, secretKey);
    byte[] decryptedData = cipher.doFinal(Base64.getDecoder().decode(encryptedFingerprint));
    return new String(decryptedData);
}
```

```
    public static void main(String[] args) {
        try {
            String fingerprint = "uniqueDeviceFingerprint"; // Örnek parmak izi

            // Parmak izini şifreleme
            String encryptedFingerprint = encryptFingerprint(fingerprint);
            System.out.println("Şifrelenmiş Parmak İzi: " + encryptedFingerprint);

            // Parmak izini çözme
            String decryptedFingerprint = decryptFingerprint(encryptedFingerprint);
            System.out.println("Çözülen Parmak İzi: " + decryptedFingerprint);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Özet:

- `encryptFingerprint` metodu, cihaz parmak izini AES ile şifreler.
- `decryptFingerprint` metodu, şifrelenmiş parmak izini çözer. Bu sayede parmak izi güvenli bir şekilde saklanır ve yalnızca yetkili kişiler tarafından erişilebilir.

Özet:

1. **Parmak İzinin Şifrelenerek Depolanması:** Cihazın benzersiz özelliklerinden parmak izi oluşturulup şifrelenir ve güvenli bir şekilde saklanır.
2. **Parmak İzi Doğrulaması:** Uygulamanın belirli bir cihazda çalışıp çalışmadığı doğrulanır.
3. **Verilerin Gizlenmesi ve Korunması:** Parmak izi verileri AES gibi güvenli algoritmalarla şifrelenir ve saldırılara karşı korunur.

2.3. Yerel Kütüphane JNI API Obfuske Etme (Native Library JNI API Obfuscation)

Teorik Açıklama: Java Native Interface (JNI) kullanılarak çağrılan yerel kütüphanelerin obfuske edilmesi, geri mühendislik işlemlerini zorlaştırır.

Uygulama Örnekleri:

1. JNI fonksiyon isimlerinin rastgele karakterlerle değiştirilmesi.
2. JNI parametrelerinin gizlenmesi ve anlaşılmasını zorlaştırma.
3. JNI hata yönetimi ile saldırganların hataları analiz etmesini engelleme.

2.3.1. JNI Fonksiyon İsimlerinin Rastgele Karakterlerle Değiştirilmesi

Açıklama:

JNI fonksiyonlarının isimleri rastgele karakterlerle değiştirilerek, yerel kütüphaneye yapılan çağrıların anlaşılmasını zorlaştırabilirsiniz. Bu, kodun geri mühendislik işlemine karşı korunmasını sağlar.

Java Kodu:

```java
public class MyJNIExample {
    // Rastgele isimlendirilmiş JNI fonksiyonu
    public native void nJx57F(); // Rastgele isimli JNI fonksiyonu

    static {
        System.loadLibrary("myNativeLib");
    }

    public static void main(String[] args) {
        MyJNIExample example = new MyJNIExample();
        example.nJx57F(); // Rastgele isimli fonksiyonu çağırma
    }
}
```

C Kodu (Yerel Kütüphane - myNativeLib.c):

```c
#include <jni.h>
#include <stdio.h>
#include "MyJNIExample.h"

// Rastgele isimlendirilmiş JNI fonksiyonu
JNIEXPORT void JNICALL Java_MyJNIExample_nJx57F(JNIEnv *env, jobject obj) {
    printf("Yerel kütüphane fonksiyonu çalıştı!\n");
}
```

Özet:

- JNI fonksiyonu `nJx57F()` gibi rastgele bir isimle tanımlanmıştır. Bu, geri mühendislik yapan bir kişinin fonksiyonun ne yaptığını anlamasını zorlaştırır.

2.3.2. JNI Parametrelerinin Gizlenmesi ve Anlaşılmasını Zorlaştırma

Açıklama:

JNI fonksiyonlarına gönderilen parametreler anlaşılmasını zorlaştıracak şekilde gizlenebilir. Örneğin, parametreler bir diziyle veya şifreli veriyle gönderilebilir.

Java Kodu:

```
public class MyJNIExample {
    // Rastgele parametre isimlendirilmiş JNI fonksiyonu
    public native void obfuscatedJNI(byte[] encryptedData);

    static {
        System.loadLibrary("myNativeLib");
    }

    public static void main(String[] args) {
        MyJNIExample example = new MyJNIExample();

        // Parametre olarak şifrelenmiş veri gönderme
        byte[] encryptedData = { 0x12, 0x34, 0x56 };
        example.obfuscatedJNI(encryptedData);
    }
}
```

C Kodu (Yerel Kütüphane - myNativeLib.c):

```c
#include <jni.h>
#include <stdio.h>
#include "MyJNIExample.h"

// Rastgele parametreli JNI fonksiyonu
JNIEXPORT void JNICALL Java_MyJNIExample_obfuscatedJNI(JNIEnv *env, jobject obj, jbyteArray encryptedData) {
    jsize length = (*env)->GetArrayLength(env, encryptedData);
    jbyte *data = (*env)->GetByteArrayElements(env, encryptedData, 0);

    printf("Veri uzunluğu: %d\n", length);

    for (int i = 0; i < length; i++) {
        printf("Veri %d: %x\n", i, data[i]);
    }

    (*env)->ReleaseByteArrayElements(env, encryptedData, data, 0);
}
```

Özet:

- JNI fonksiyonu bir byte dizisi alır ve veriler şifrelenmiş veya anlamsız şekilde saklanarak fonksiyonun ne yaptığı daha zor anlaşılır hale getirilir.

2.3.3. JNI Hata Yönetimi ile Saldırganların Hataları Analiz Etmesini Engelleme

Açıklama:

JNI fonksiyonlarında meydana gelen hatalar saldırganlar tarafından analiz edilmemelidir.

Bunun için özel hata yönetimi uygulanır ve detaylı hata mesajları saklanır.

Java Kodu:

```
public class MyJNIExample {
    public native void performTask();

    static {
        System.loadLibrary("myNativeLib");
    }

    public static void main(String[] args) {
        MyJNIExample example = new MyJNIExample();
        try {
            example.performTask();
        } catch (Exception e) {
            System.out.println("Bir hata oluştu.");
        }
    }
}
```

C Kodu (Yerel Kütüphane - myNativeLib.c):

```c
#include <jni.h>
#include <stdio.h>
#include "MyJNIExample.h"

// Özel hata yönetimi
JNIEXPORT void JNICALL Java_MyJNIExample_performTask(JNIEnv *env, jobject obj) {
    // Hata mesajı detaysız
    if (/* Bir hata meydana gelirse */) {
        printf("Bir hata meydana geldi\n");
        return;
    }

    // Normal işlem
    printf("Görev başarıyla tamamlandı.\n");
}
```

Özet:

- Hata meydana geldiğinde yalnızca genel bir mesaj ("Bir hata meydana geldi") gösterilir. Bu, saldırganların hatalardan fazla bilgi edinmesini engeller.

Özet:

1. **JNI Fonksiyon İsimlerinin Rastgeleleştirilmesi:** Fonksiyon isimlerini rastgele karakterlerle değiştirerek analiz edilmeyi zorlaştırabilirsiniz.
2. **JNI Parametrelerinin Gizlenmesi:** Parametreler şifrelenmiş veya gizlenmiş formatta gönderilerek ne yapıldığını anlamayı zorlaştırabilirsiniz.
3. **JNI Hata Yönetimi:** Hatalar detaylı mesajlarla paylaşılmaz, bu da saldırganların analizine karşı koruma sağlar.

2.4. Statik Dizelerin Obfuske Edilmesi (Obfuscation of Static Strings)

Teorik Açıklama: Statik dizeler, saldırganların geri mühendislik işlemleri sırasında kullanabileceği önemli bilgiler içerir. Bu dizelerin obfuske edilmesi, güvenliği artırır.

Uygulama Örnekleri:

1. Statik dizelerin şifrelenmesi ve çalışma anında çözülmesi.
2. Dizelerin obfuske edilerek anlamlarının gizlenmesi.
3. Rastgele dize oluşturma ve manipülasyon teknikleri ile güvenliği artırma.

2.4.1. Statik Dizelerin Şifrelenmesi ve Çalışma Anında Çözülmesi

Açıklama:

Statik dizeler genellikle önemli bilgiler içerir (örneğin, API anahtarları veya kullanıcı bilgileri) ve geri mühendislik yapan bir saldırgan bu bilgilere kolayca erişebilir. Bu nedenle, statik dizeler şifrelenir ve çalışma anında çözülerek güvenlik artırılır.

Java Kodu:

```
import javax.crypto.Cipher;
import javax.crypto.spec.SecretKeySpec;
import java.util.Base64;

public class StaticStringObfuscation {

    private static final String KEY = "1234567890123456"; // Şifreleme anahtarı (16-byte AES)

    // Statik dizenin şifrelenmiş hali (örneğin bir API anahtarı)
    private static final String ENCRYPTED_STRING = "Y2hlY2tfdGhpcyBzdHJpbmdf";
```

```
// Şifreleme fonksiyonu (dizeleri şifrelemek için)
public static String encrypt(String data) throws Exception {
    SecretKeySpec secretKey = new SecretKeySpec(KEY.getBytes(), "AES");
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.ENCRYPT_MODE, secretKey);
    byte[] encryptedBytes = cipher.doFinal(data.getBytes());
    return Base64.getEncoder().encodeToString(encryptedBytes);
}
```

```
// Şifre çözme fonksiyonu
public static String decrypt(String encryptedData) throws Exception {
    SecretKeySpec secretKey = new SecretKeySpec(KEY.getBytes(), "AES");
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(Cipher.DECRYPT_MODE, secretKey);
    byte[] decodedBytes = Base64.getDecoder().decode(encryptedData);
    byte[] decryptedBytes = cipher.doFinal(decodedBytes);
    return new String(decryptedBytes);
}
```

```
    public static void main(String[] args) {
        try {
            // Statik dizenin şifresini çözme
            String decryptedString = decrypt(ENCRYPTED_STRING);
            System.out.println("Çözülen dize: " + decryptedString);

        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}
```

Özet:

- `ENCRYPTED_STRING` değişkeni şifrelenmiş bir statik dizeyi temsil eder. Program çalışırken bu şifre çözülerek gerçek veri elde edilir.
- Şifreleme AES algoritmasıyla yapılmıştır ve çözüm Base64 kodlaması ile gerçekleştirilmiştir.

2.4.2. Dizelerin Obfuske Edilerek Anlamlarının Gizlenmesi

Açıklama:

Dizelerin doğrudan kodda yer alması, bu dizelerin anlamlarını açık eder. Dizeleri obfuske etmek, saldırganların anlamlarını anlamalarını zorlaştırır.

Java Kodu:

```
public class StaticStringObfuscation {

    // Obfuske edilmiş dizeler
    private static final char[] OBFUSCATED_STRING = { 'J', 'a', 'v', 'a', 'I', 's', 'S', 'e', 'c', 'u', 'r', 'e' };

    // Dizeyi çözümleyerek gerçek değerini elde etme
    public static String decodeObfuscatedString() {
        StringBuilder decodedString = new StringBuilder();
        for (char c : OBFUSCATED_STRING) {
            decodedString.append(c);
        }
        return decodedString.toString();
    }

    public static void main(String[] args) {
        // Obfuske edilmiş dizeyi çözme ve gösterme
        String decoded = decodeObfuscatedString();
        System.out.println("Çözülen dize: " + decoded);
    }
}
```

Açıklama:

- `OBFUSCATED_STRING` dizisi, dizeyi karakter karakter saklayarak doğrudan görünmesini engeller.
- `decodeObfuscatedString()` metodu, obfuske edilmiş dizeyi çözer ve anlamlı hale getirir. Bu yaklaşım, dizelerin açıkça görünmemesini sağlar.

2.4.3. Rastgele Dize Oluşturma ve Manipülasyon Teknikleri ile Güvenliği Artırma

Açıklama:

Rastgele dize oluşturma ve bu dizeleri çalışma anında manipüle etme, dizelerin açıkça kodda görünmesini engeller ve analiz edilmesini zorlaştırır.

Java Kodu:

```java
import java.util.Random;

public class RandomStringObfuscation {

    // Rastgele bir dize oluşturma fonksiyonu
    public static String generateRandomString(int length) {
        String characters = "ABCDEFGHIJKLMNOPQRSTUVWXYZabcdefghijklmnopqrstuvwxyz0123456789";
        StringBuilder randomString = new StringBuilder();
        Random random = new Random();

        for (int i = 0; i < length; i++) {
            randomString.append(characters.charAt(random.nextInt(characters.length())));
        }

        return randomString.toString();
    }
```

```
    // Manipüle edilen dizeyi elde etme
    public static String manipulateString(String input) {
        return input.substring(0, input.length() / 2); // Dizenin yarısını kullan
    }
```

```
    public static void main(String[] args) {
        // Rastgele bir dize oluşturma
        String randomString = generateRandomString(16);
        System.out.println("Oluşturulan rastgele dize: " + randomString);

        // Dizeyi manipüle etme
        String manipulatedString = manipulateString(randomString);
        System.out.println("Manipüle edilmiş dize: " + manipulatedString);
    }
}
```

Özet:

- `generateRandomString()` metodu, rastgele bir dize oluşturur.
- `manipulateString()` metodu bu diziyi manipüle eder. Bu teknik, sabit bir dize yerine dinamik ve manipüle edilmiş bir dize kullanarak dizelerin analiz edilmesini zorlaştırır.

Özet:

1. **Statik Dizelerin Şifrelenmesi ve Çalışma Anında Çözülmesi:** Statik dizeler şifrelenerek saklanır ve çalışma anında çözülür.
2. **Dizelerin Obfuske Edilerek Gizlenmesi:** Dizeler karakter dizisi olarak saklanır ve çalışma anında çözülerek gizlenir.
3. **Rastgele Dize Oluşturma ve Manipülasyon Teknikleri:** Rastgele oluşturulmuş dizeler, statik olmaktan çıkar ve manipüle edilerek güvenlik artırılır.

Haftanın Özeti ve Gelecek Hafta

Bu Hafta:

- Kod Güçlendirme Teknikleri (C/C++ ve Java)
- Obfuske Teknikleri ve Uygulamaları

Gelecek Hafta:

- Saldırı Ağaçları ve Güvenlik Modelleri
- Saldırı Yöntemleri ve Güvenli İletişim

$4. Hafta - Sonu$