1.1.1 Depolu Komut Argümanı Enjeksiyonu (Stored Command Argument Injection) (CWE-88)

Açıklık Önem Derecesi: Düşük

Açıklığın Etkisi: Komut çalıştırma, tampon taşırma, dizin gezinme, kod çalıştırma

Açıklığın Barındıran Dosyalar/Satırlar:

Proje Dosyası/Dosya Adı	Satır Numarası

Açıklığın Açıklaması:

Geliştiriciler uygulamalarında kabuk (shell) komutu çalıştırmak için (yani uygulamalarının dışında harici bir program çalıştırmak için) komutları ve parametre & argümanlarını string (dize) şeklinde tanımlarlar. Bu string'ler statik olabileceği gibi dinamik de olabilmektedir. Eğer string dahili veya harici argüman almıyorsa statik, alıyorsa dinamiktir. Geliştiriciler bu string'lere argüman olarak harici veya dahili girdi ekleyebilmektedirler. Böylesi bir uygulamada geliştirici kabuk komutu ile kendi tanımladığı argümanların çalışacağını varsayar. Ancak string'e konulan girdi kötü niyetli ellerden çıkmış bir değer ise geliştiricinin varsaydığı çalışacak argümanlar dışında daha fazla sayıda argüman çalışabilir. Girdiler kabuk komutuna (harici bir programa) argüman olarak ilave edildiğinde filtreleme yapılmıyorsa "Komut Argümanı Enjeksiyonu (CWE-88)" açıklığı meydana gelir. Eğer girdiler sistemin bir bileşeninden geliyorsa "Depolu Komut Argümanı Enjeksiyonu (CWE-88)" açıklığı meydana gelir.

Saldırgan kontrollü argümanlar ile yapılabilecekler kabuk (shell) komutunun işlevselliğine, kapasitesine, uygulanışına ve komuta verilmiş izin seviyesine bağlıdır. Örneğin;

- Uygulama kaynak kodlarında kabuk komutu (harici program) çağrıldığında "Command Injection"
- Uygulama kaynak kodlarında kabuk komutu (harici program) olarak derleyici ya da yorumlayıcı (interpreter) çağırıldığında "Code Injection"
- Uygulama kaynak kodlarında kabuk komutu (harici program) dosya yollarını düzenlemenin mümkün olduğu argüman yapısında olduğunda ve çağırıldığında "Path Traversal"
- Kabuk komutu kendi içerisinde açıklık barındırdığında ve çağırıldığında "Buffer Overflow"

zafiyetleri meydana gelebilir.

Depolu Komut Argümanı Enjeksiyonu açıklığını izah etmek adına Java ve PHP dillerinde güvensiz ve güvenli kod blokları örneklerine yer verilmiştir:

Java - Güvensiz Kod Örneği:

```
// GÜVENSİZ ÖRNEK

// ENG: Execute External Program with Arguments from Database
//
// TUR: Veri Tabanından Çekilen Argümanlar ile Harici Bir
//      Program Çalıştırma

private String executeSystemCommand_Unsafe(HttpServletRequest request) throws
Exception {
    String commandResult;

    String param = request.getParameter("CommandID");
    int commandID = Integer.parseInt(param);

    ResultSet rs = readCommandFromDB(commandID);
    String commandFromDB = rs.getString("Arg");

    try {
        Runtime runtime = Runtime.getRuntime();
        Process subProc = runtime.exec(EXTERNAL_PROGRAM + " " +
commandFromDB);

        BufferedReader irProcOutput = new BufferedReader(new
InputStreamReader(subProc.getInputStream()));

        String line = null;
        while ((line = irProcOutput.readLine()) != null)
            commandResult += line;

        irProcOutput.close();

    } catch (Exception ex) {
        handleExceptions(ex);
    }

    return commandResult;
}
```

Güvensiz kod bloğu örneğinde veri tabanından çekilen bir girdi filtrelenmeden çalıştırılacak kabuk (shell) komutunun sonuna eklenmektedir. Veri tabanındaki girdi daha önceden

saldırganca oluşturulmuşsa veya saldırganca manipule edilmişse komut bu haliyle çalıştığında saldırganın girdileri mevcut argümanların sonuna enjekte olabilir. Girdi kontrolü eksikliği nedeniyle buradaki kullanım güvensizdir.

Java - Güvenli Kod Örneği:

```
// GÜVENLİ ÖRNEK

// ENG: Execute External Program with Alphanumeric
//      Arguments from DB, Sanitized with Regex
//
// TUR: Regexp ile Filtrelenmiş, Veri Tabanından
//      Çekilmiş Alfanümerk Argumanlar ile Harici
//      Program Çalıştırma

private String executeSystemCommand_Safe(HttpServletRequest request) throws
Exception {
    String commandResult;

    String param = request.getParameter("CommandID");
    int commandID = Integer.parseInt(param);

    ResultSet rs = readCommandFromDB(commandID);
    String commandFromDB = rs.getString("Arg");
    commandFromDB = commandFromDB.replaceAll("[^A-Za-z0-9]", "");
    String[] programArgs = new String[] ( EXTERNAL_EXECUTABLE, commandFromDB
);
    try {
        Runtime runtime = Runtime.getRuntime();
        Process subProc = runtime.exec(programArgs );

        BufferedReader irProcOutput = new BufferedReader(new
InputStreamReader(subProc.getInputStream()));

        String line = null;
        while ((line = irProcOutput.readLine()) != null)
            commandResult += line;

        irProcOutput.close();

    } catch (Exception ex) {
        handleExceptions(ex);
    }

    return commandResult;
}
```

Güvenli kod bloğu örneğinde ise veri tabanından çekilen bir girdi regular expression denetlemesine tabi tutulmaktadır. Bu denetlemeye göre A'dan Z'ye, a'dan z'ye ve 0'dan 9'a kadarki karakterlerden olmayan her karakter silinmektedir. Bu şekilde bir filtrelemenin var olması mevcut argümanların sonunda saldırgan kontrollü argüman enjeksiyonunun yapılamayacağını gösterir. Bu kullanım güvenli kabul edilir.

PHP - Güvensiz Kod Bloğu:

```php
// GÜVENSİZ KOD

<?php
    $result = mysql_query($conn, $sql);                // Taint Source (Input)

    if (mysql_num_rows($result) > 0) {

        $row = mysql_fetch_assoc($result);
        $filename = $row["filename"];
        $cmd = "find . -type f -name ";

        system(escapeshellcmd($cmd . $filename));      // Taint Sink

    }
?>
```

PHP - Güvenli Kod Bloğu:

```php
// GÜVENLİ KOD

<?php
    $result = mysql_query($conn, $sql);                // Taint Source (Input)

    if (mysql_num_rows($result) > 0) {

        $row = mysql_fetch_assoc($result);
        $filename = $row["filename"];
        $cmd = "find . -type f -name ";

        system(escapeshellcmd($cmd . $filename));      // Taint Sink

    }
?>
```

Güvensiz kod bloğu örneğinde veri tabanından girdi çekiliyor ve bu girdi olduğu gibi shell komut çalıştırma fonksiyonundaki komuta argüman olarak ekleniyor. Burada filtreleme

olmaması ve doğrudan yerleştirme olması sebebiyle Depolu Komut Argümanı Enjeksiyonu açıklığı mevcuttur.

Güvenli kod bloğu örneğinde ise veri tabanından girdi çekilmektedir ve bu girdi regular expression ile denetime tabi tutulmaktadır. Bu denetlemeye göre A'dan Z'ye, a'dan z'ye ve 0'dan 9'a kadarki karakterlerden olmayan her karakter silinmektedir. Böylece depolu komut argümanı enjeksiyonunda kullanılabilecek karakterlerden girdi ayıklandığından depolu komut argümanı enjeksiyonu ihtimali ortadan kaldırılmış oluyor. Bu güvenli kabul edilen kullanımdır.

Ekstra olarak depolu komut argümanı enjeksiyonuna java'dan şu örnek de verilebilir:

Java - Güvensiz Kod Bloğu Örneği (2):

```
// GÜVENSİZ ÖRNEK

// ENG: Execute External Program with Arguments from Environment
//
// TUR: Shell Ortamından Çekilen Argümanlar ile Harici Bir
//      Program Çalıştırma

private String executeSystemCommand_Unsafe(HttpServletRequest request) throws
Exception {
    String commandResult;

    String param = os.getEnv("CommandID");

    try {
        Runtime runtime = Runtime.getRuntime();
        Process subProc = runtime.exec(EXTERNAL_PROGRAM + " " + param);

        BufferedReader irProcOutput = new BufferedReader(new
InputStreamReader(subProc.getInputStream()));

        String line = null;
        while ((line = irProcOutput.readLine()) != null)
            commandResult += line;

        irProcOutput.close();

    } catch (Exception ex) {
        handleExceptions(ex);
    }

    return commandResult;
}
```

Bu güvensiz örnekte java kaynak kodları sistemin environment variable'ını (ortam değişkenini) çekmektedir ve bunları sistem shell (kabuk) komutuna doğrudan argüman olarak eklemektedir. Eğer uygulamada gelecekte environment variable injection zafiyetine sahip bir bileşen ortaya çıkarsa (veya zafiyet halihazırda zaten varsa) environment variable enjeksiyon yoluyla saldırganlar sistemdeki ortam değişkenlerinin değerlerini değiştirebilirler. Uygulama sistemin kabuğundan bu manipule edilmiş ortam değişkenini çekip komut sonrasına eklediğinde ise mevcut kabuk komutlarındaki argümanlara ilave argümanlar şeklinde eklenebilir. Yani komut argümanı enjeksiyonu saldırısı yaşanabilir.

Java - Güvenli Kod Bloğu Örneği (2):

```
// GÜVENLİ ÖRNEK

// ENG: Execute External Program with Alphanumeric
//      Arguments from Environment, Sanitized with Regex
//
// TUR: Regexp ile Filtrelenmiş, Environment'tan
//      Çekilmiş Alfanümerk Argumanlar ile Harici
//      Program Çalıştırma

private String executeSystemCommand_Unsafe(HttpServletRequest request) throws
Exception {
    String commandResult;

    String param = os.getEnv("arguments");
    param = param.replaceAll("[^A-Za-z0-9]", "");
    String[] programArgs = new String[] ( EXTERNAL_EXECUTABLE, param );

try {
        Runtime runtime = Runtime.getRuntime();
        Process subProc = runtime.exec( programArgs );

        BufferedReader irProcOutput = new BufferedReader(new
InputStreamReader(subProc.getInputStream()));

        String line = null;
        while ((line = irProcOutput.readLine()) != null)
            commandResult += line;

        irProcOutput.close();

    } catch (Exception ex) {
        handleExceptions(ex);
    }

    return commandResult;
}
```

Bu güvenli örnekte ise java kaynak kodları sistemin environment variable'ını (ortam değişkenini) yine çekmektedir, fakat bu sefer regular expression ile denetime tabi tuttuktan sonra sistem shell (kabuk) komutuna argüman olarak eklemektedir. Eğer uygulamada gelecekte environment variable injection zafiyetine sahip bir bileşen ortaya çıkarsa (veya zafiyet halihazırda zaten varsa) environment variable enjeksiyon yoluyla saldırganlar sistemdeki ortam değişkenleri değerlerini değiştirebilirler ama komut argümanı enjeksiyonu karakterlerine sahip karakterler eklemiş olsalar da uygulamanın mevcut kabuk

komutlarındaki argümanlara ilave argümanlar ekleyemezler. Çünkü komuta argümanlar eklenirken filtrelenmektedir. Böylece ortam değişkeni enjeksiyonu çıksa bile komut argümanı enjeksiyonu zafiyeti önlenmiş olur.

Kurum uygulamada Stored Command Argument Injection (CWE-88) açıklığı tespit edilmiştir:

::::::::::::::::BULGU::::::::::::::::::::

Şekil XXX. Girdi Doğrulaması Yapılmamış Argüman

Açıklığın Önlemi:

Tavsiyeler şu şekildedir:

- Kaynak kodda herhangi bir kabuk (shell) komutunu doğrudan çalıştırmak yerine API'lerin veya kütüphanelerin sunduğu çözümler tercih edilmelidir.
- Kaynak kodda eğer doğrudan kabuk (shell) komutu çalıştırmak zaruri durumda ise yalnızca dinamik girdinin dahil edilmediği statik komutlar kullanmak tercih edilmelidir.
- Tüm girdiler kaynağı neresi olursa olsun doğrulamadan geçirilmelidir. Bu doğrulama belirli yapıdaki girdileri reddetme işlemi yerine sadece belirli yapıya uyanların kabul edildiği şeklinde olmalıdır. Yani whitelist (beyaz liste) kullanılmalıdır. Siyah liste (blacklist) önlemi kullanılmamalıdır. Bir girdiyi doğrulamada şunlar kontrol edilebilir:
    - Veri türü (int, string, float, byte, … v.b. tipte mi geliyor kontrolü)
    - Boyutu (x KB, y MB,…v.b. boyutta mı geliyor kontrolü)
    - Aralığı (Veri ölçülebilir bir sayısal değerse aralık sınırlarını aşıyor mu kontrolü)
    - Formatı (çözümlenebilir şifreli, tek yönlü şifreli, açık metin, … v.b. biçimde mi geliyor kontrolü)
    - Beklenen Değerlerden Biri Olup Olmadığı (Whitelist'te tanımlı desenlerden biri mi geliyor kontrolü)
- Derinlikli defans (defense-in-depth) prensibi gereği gelecekte yaşanabilir ihtimaline binaen hasarı şimdiden minimize etmek için uygulamalar fuzuli işletim sistemi izinlerine sahip olmayan kısıtlı modlardaki kullanıcı hesapları kullanacak şekilde yapılandırılmalıdır.
- Eğer mümkünse en az ayrıcalık prensibi (principle of least privilege) gereği uygulama tarafından kullanılan spesifik komutlar ve dosyalar için minimum izinlere sahip, özel bir kullanıcı hesabı kullanılmalıdır ve böylece tüm işletim sistemi seviyesindeki komutlar izole edilmelidir.

Referanslar:

1. https://cwe.mitre.org/data/definitions/88.html
2.