Article • 08/13/201516 minutes to read

November 2009

Volume 24 Number 11

Security Briefs - XML Denial of Service Attacks and Defenses

By Bryan Sullivan | November 2009

Denial of service (DoS) attacks are among the oldest types of attacks against Web sites. Documented DoS attacks exist at least as far back as 1992, which predates SQL injection (discovered in 1998), cross-site scripting (JavaScript wasn't invented until 1995), and cross-site request forgery (CSRF attacks generally require session cookies, and cookies weren't introduced until 1994).

From the beginning, DoS attacks were highly popular with the hacker community, and it's easy to understand why. A single "script kiddie" attacker with a minimal amount of skill and resources could generate a flood of TCP SYN (for synchronize) requests sufficient to knock a site out of service. For the fledgling e-commerce world, this was devastating: if users couldn't get to a site, they couldn't very well spend money there either. DoS attacks were the virtual equivalent of erecting a razor-wire fence around a brick-and-mortar store, except that any store could be attacked at any time, day or night.

Over the years, SYN flood attacks have been largely mitigated by improvements in Web server software and network hardware. However, lately there has been a resurgence of interest in DoS attacks within the security community—not for "old school" network-level DoS, but instead for application-level DoS and particularly for XML parser DoS.

XML DoS attacks are extremely asymmetric: to deliver the attack payload, an attacker needs to spend only a fraction of the processing power or bandwidth that the victim needs to spend to handle the payload. Worse still, DoS vulnerabilities in code that processes XML are also extremely widespread. Even if you're using thoroughly tested parsers like those found in the Microsoft .NET Framework System.Xml classes, your code can still be vulnerable unless you take explicit steps to protect it.

This article describes some of the new XML DoS attacks. It also shows ways for you to detect potential DoS vulnerabilities and how to mitigate them in your code.

XML Bombs

One type of especially nasty XML DoS attack is the XML bomb—a block of XML that is both well-formed and valid according to the rules of an XML schema but which crashes or hangs a program when that program attempts to parse it. The best-known example of an XML bomb is probably the Exponential Entity Expansion attack.

Inside an XML document type definition (DTD), you can define your own entities, which essentially act as string substitution macros. For example, you could add this line to your DTD to replace all occurrences of the string &companyname; with "Contoso Inc.":

```
<!ENTITY companyname "Contoso Inc.">
```

You can also nest entities, like this:

```
<!ENTITY companyname "Contoso Inc.">
<!ENTITY divisionname "&companyname; Web Products Division">
```

While most developers are familiar with using external DTD files, it's also possible to include inline DTDs along with the XML data itself. You simply define the DTD directly in the <!DOCTYPE > declaration instead of using <!DOCTYPE> to refer to an external DTD file:

```
<?xml version="1.0"?>
<!DOCTYPE employees [
  <!ELEMENT employees (employee)*>
  <!ELEMENT employee (#PCDATA)>
  <!ENTITY companyname "Contoso Inc.">
  <!ENTITY divisionname "&companyname; Web Products Division">
]>
<employees>
  <employee>Glenn P, &divisionname;</employee>
  <employee>Dave L, &divisionname;</employee>
</employees>
```

An attacker can now take advantage of these three properties of XML (substitution entities, nested entities, and inline DTDs) to craft a malicious XML bomb. The attacker writes an XML document with nested entities just like the previous example, but instead of nesting just one level deep, he nests his entities many levels deep, as shown here:

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
  <!ENTITY lol "lol">
  <!ENTITY lol2 "&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;&lol;">
  <!ENTITY lol3 "&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;&lol2;">
  <!ENTITY lol4 "&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;&lol3;">
  <!ENTITY lol5 "&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;&lol4;">
  <!ENTITY lol6 "&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;&lol5;">
  <!ENTITY lol7 "&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;&lol6;">
  <!ENTITY lol8 "&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;&lol7;">
  <!ENTITY lol9 "&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;&lol8;">
]>
<lolz>&lol9;</lolz>
```

It should be noted that this XML is both well-formed and valid according to the rules of the DTD. When an XML parser loads this document, it sees that it includes one root element, "lolz", that contains the text "&lol9;". However, "&lol9;" is a defined entity that expands to a string containing ten "&lol8;" strings. Each "&lol8;" string is a defined entity that expands to ten "&lol7;" strings, and so forth. After all the entity expansions have been processed, this small (< 1 KB) block of XML will actually contain a billion "lol"s, taking up almost 3GB of memory! You can try this attack (sometimes called the Billion Laughs attack) for yourself using this very simple block of code—just be prepared to kill your test app process from Task Manager:

```
void processXml(string xml)
{
    System.Xml.XmlDocument document = new XmlDocument();
    document.LoadXml(xml);
}
```

Some of the more devious readers may be wondering at this point whether it's possible to create an infinitely recursing entity expansion consisting of two entities that refer to each other:

```
<?xml version="1.0"?>
<!DOCTYPE lolz [
  <!ENTITY lol1 "&lol2;">
  <!ENTITY lol2 "&lol1;">
]>
<lolz>&lol1;</lolz>
```

This would be a very effective attack, but fortunately it isn't legal XML and will not parse. However, another variation of the Exponential Entity Expansion XML bomb that does work is the Quadratic Blowup attack, discovered by Amit Klein of Trusteer. Instead of defining multiple small, deeply nested entities, the attacker defines one very large entity and refers to it many times:

```
<?xml version="1.0"?>
<!DOCTYPE kaboom [
  <!ENTITY a "aaaaaaaaaaaaaaaaaa...">
]>
<kaboom>&a;&a;&a;&a;&a;&a;&a;&a;&a;...</kaboom>
```

If an attacker defines the entity "&a;" as 50,000 characters long, and refers to that entity 50,000 times inside the root "kaboom" element, he ends up with an XML bomb attack payload slightly over 200 KB in size that expands to 2.5 GB when parsed. This expansion ratio is not quite as impressive as with the Exponential Entity Expansion attack, but it is still enough to take down the parsing process.

Another of Klein's XML bomb discoveries is the Attribute Blowup attack. Many older XML parsers, including those in the .NET Framework versions 1.0 and 1.1, parse XML attributes in an extremely inefficient quadratic O(n^2) runtime. By creating an XML document with a large number of attributes (say 100,000 or more) for a single element, the XML parser will monopolize the processor for a long period of time and therefore cause a denial of service condition. However, this vulnerability has been fixed in .NET Framework versions 2.0 and later.

External Entity Attacks

Instead of defining entity replacement strings as constants, it is also possible to define them so that their values are pulled from external URIs:

```
<!ENTITY stockprice SYSTEM    "https://www.contoso.com/currentstockprice.ashx">
```

While the exact behavior depends on the particular XML parser implementation, the intent here is that every time the XML parser encounters the entity "&stockprice;" the parser will make a request to www.contoso.com/currentstockprice.ashx and then substitute the response received from that request for the stockprice entity. This is undoubtedly a cool and useful feature of XML, but it also enables some devious DoS attacks.

The simplest way to abuse the external entity functionality is to send the XML parser to a resource that will never return; that is, to send it into an infinite wait loop. For example, if an attacker had control of the server adatum.com, he could set up a generic HTTP handler file at https://adatum.com/dos.ashx as follows:

```
using System;
using System.Web;
using System.Threading;

public class DoS : IHttpHandler
{
    public void ProcessRequest(HttpContext context)
    {
        Thread.Sleep(Timeout.Infinite);
    }

    public bool IsReusable { get { return false; } }
}
```

He could then craft a malicious entity that pointed to https://adatum.com/dos.ashx , and when the XML parser reads the XML file, the parser would hang. However, this is not an especially effective attack. The point of a DoS attack is to consume resources so that they are unavailable to legitimate users of the application. Our earlier examples of Exponential Entity Expansion and Quadratic Blowup XML bombs caused the server to use large amounts of memory and CPU time, but this example does not. All this attack really consumes is a single thread of execution. Let's improve this attack (from the attacker's perspective) by forcing the server to consume some resources:

```
public void ProcessRequest(HttpContext context)
{
    context.Response.ContentType = "text/plain";
    byte[] data = new byte[1000000];
    for (int i = 0; i < data.Length; i++) { data[i] = (byte)'A'; }
    while (true)
    {
        context.Response.OutputStream.Write(data, 0, data.Length);
        context.Response.Flush();
    }
}
```

This code will write an infinite number of ‘A’ characters (one million at a time) to the response stream and chew up a huge amount of memory in a very short amount of time. If the attacker is unable or unwilling to set up a page of his own for this purpose—perhaps he doesn’t want to leave a trail of evidence that points back to him—he can instead point the external entity to a very large resource on a third-party Web site. Movie or file downloads can be especially effective for this purpose; for example, the Visual Studio 2010 Professional beta download is more than 2GB.

Yet another clever variation of this attack is to point an external entity at a target server’s own intranet resources. Discovery of this attack technique is credited to Steve Orrin of Intel. This technique does require the attacker to have internal knowledge of intranet sites accessible by the server, but if an intranet resource attack can be executed, it can be especially effective because the server is spending its own resources (processor time, bandwidth, and memory) to attack itself or its sibling servers on the same network.

Defending Against XML Bombs

The easiest way to defend against all types of XML entity attacks is to simply disable altogether the use of inline DTD schemas in your XML parsing objects. This is a straightforward application of the principle of attack surface reduction: if you’re not using a feature, turn it off so that attackers won’t be able to abuse it.

In .NET Framework versions 3.5 and earlier, DTD parsing behavior is controlled by the Boolean ProhibitDtd property found in the System.Xml.XmlTextReader and System.Xml.XmlReaderSettings classes. Set this value to true to disable inline DTDs completely:

```
XmlTextReader reader = new XmlTextReader(stream);
reader.ProhibitDtd = true;
```

or

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ProhibitDtd = true;
XmlReader reader = XmlReader.Create(stream, settings);
```

The default value of ProhibitDtd in XmlReaderSettings is true, but the default value of ProhibitDtd in XmlTextReader is false, which means that you have to explicitly set it to true to disable inline DTDs.

In .NET Framework version 4.0 (in beta at the time of this writing), DTD parsing behavior has been changed. The ProhibitDtd property has been deprecated in favor of the new DtdProcessing property. You can set this property to Prohibit (the default value) to cause the runtime to throw an exception if a <!DOCTYPE> element is present in the XML:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.DtdProcessing = DtdProcessing.Prohibit;
XmlReader reader = XmlReader.Create(stream, settings);
```

Alternatively, you can set the DtdProcessing property to Ignore, which will not throw an exception on encountering a <!DOCTYPE> element but will simply skip over it and not process it. Finally, you can set DtdProcessing to Parse if you do want to allow and process inline DTDs.

If you really do want to parse DTDs, you should take some additional steps to protect your code. The first step is to limit the size of expanded entities. Remember that the attacks I’ve discussed work by creating entities that expand to huge strings and force the parser to consume large amounts of memory. By setting the MaxCharactersFromEntities property of the XmlReaderSettings object, you can cap the number of characters that can be created through entity expansions. Determine a reasonable maximum and set the property accordingly. Here’s an example:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ProhibitDtd = false;
settings.MaxCharactersFromEntities = 1024;
XmlReader reader = XmlReader.Create(stream, settings);
```

Defending Against External Entity Attacks

At this point, we have hardened this code so that it is much less vulnerable to XML bombs, but we haven’t yet addressed the dangers posed by malicious external entities. You can improve your resilience against these attacks if you customize the behavior of XmlReader by changing its XmlResolver. XmlResolver objects are used to resolve external references, including external entities. XmlTextReader instances, as well as XmlReader instances returned from calls to XmlReader.Create, are prepopulated with default XmlResolvers (actually XmlUrlResolvers). You can prevent XmlReader from resolving external entities while still allowing it to resolve inline entities by setting the XmlResolver property of XmlReaderSettings to null. This is attack surface reduction at work again; if you don’t need the capability, turn it off:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.ProhibitDtd = false;
settings.MaxCharactersFromEntities = 1024;
settings.XmlResolver = null;
XmlReader reader = XmlReader.Create(stream, settings);
```

If this situation doesn’t apply to you—if you really, truly need to resolve external entities—all hope is not lost, but you do have a little more work to do. To make XmlResolver more resilient to denial of service attacks, you need to change its behavior in three ways. First, you need to set a request timeout to prevent infinite delay attacks. Second, you need to limit the amount of data that it will retrieve. Finally, as a defense-in-depth measure, you need to restrict the XmlResolver from retrieving resources on the local host. You can do all of this by creating a custom XmlResolver class.

The behavior that you want to modify is governed by the XmlResolver method GetEntity. Create a new class XmlSafeResolver derived from XmlUrlResolver and override the GetEntity method as follows:

```
class XmlSafeResolver : XmlUrlResolver
{
    public override object GetEntity(Uri absoluteUri, string role,
        Type ofObjectToReturn)
    {

    }
}
```

The default behavior of the XmlUrlResolver.GetEntity method looks something like the following code, which you can use as a starting point for your implementation:

```
public override object GetEntity(Uri absoluteUri, string role,
    Type ofObjectToReturn)
{
    System.Net.WebRequest request = WebRequest.Create(absoluteUri);
    System.Net.WebResponse response = request.GetResponse();
    return response.GetResponseStream();
}
```

The first change is to apply timeout values when making the request and when reading the response. Both the System.Net.WebRequest and the System.IO.Stream classes provide inherent support for timeouts. In the sample code shown in **Figure 1**, I simply hardcode the timeout value, but you could easily expose a public Timeout property on the XmlSafeResolver class if you want greater configurability.

Figure 1 Configuring Timeout Values

```
private const int TIMEOUT = 10000;  // 10 seconds

public override object GetEntity(Uri absoluteUri, string role,
   Type ofObjectToReturn)
{
    System.Net.WebRequest request = WebRequest.Create(absoluteUri);
    request.Timeout = TIMEOUT;

    System.Net.WebResponse response = request.GetResponse();
    if (response == null)
        throw new XmlException("Could not resolve external entity");

    Stream responseStream = response.GetResponseStream();
    if (responseStream == null)
        throw new XmlException("Could not resolve external entity");
    responseStream.ReadTimeout = TIMEOUT;
    return responseStream;
}
```

The next step is to cap the maximum amount of data that is retrieved in the response. There's no "MaxSize" property or the equivalent for the Stream class, so you have to implement this functionality yourself. To do this, you can read data from the response stream one chunk at a time and copy it into a local stream cache. If the total number of bytes read from the response stream exceeds a predefined limit (again hardcoded for simplicity only), you stop reading from the stream and throw an exception (see **Figure 2**).

Figure 2 Capping the Maximum Amount of Data Retrieved

```
private const int TIMEOUT = 10000;                   // 10 seconds
private const int BUFFER_SIZE = 1024;                // 1 KB
private const int MAX_RESPONSE_SIZE = 1024 * 1024;   // 1 MB

public override object GetEntity(Uri absoluteUri, string role,
   Type ofObjectToReturn)
{
    System.Net.WebRequest request = WebRequest.Create(absoluteUri);
    request.Timeout = TIMEOUT;

    System.Net.WebResponse response = request.GetResponse();
    if (response == null)
        throw new XmlException("Could not resolve external entity");

    Stream responseStream = response.GetResponseStream();
    if (responseStream == null)
        throw new XmlException("Could not resolve external entity");
    responseStream.ReadTimeout = TIMEOUT;

    MemoryStream copyStream = new MemoryStream();
    byte[] buffer = new byte[BUFFER_SIZE];
    int bytesRead = 0;
    int totalBytesRead = 0;
    do
    {
        bytesRead = responseStream.Read(buffer, 0, buffer.Length);
        totalBytesRead += bytesRead;
        if (totalBytesRead > MAX_RESPONSE_SIZE)
            throw new XmlException("Could not resolve external entity");
        copyStream.Write(buffer, 0, bytesRead);
    } while (bytesRead > 0);

    copyStream.Seek(0, SeekOrigin.Begin);
    return copyStream;
}
```

As an alternative, you can wrap the Stream class and implement the limit checking directly in the overridden Read method (see **Figure 3**). This is a more efficient implementation since you save the extra memory allocated for the cached MemoryStream in the earlier example.

Figure 3 Defining a Size-Limited Stream Wrapper Class

```
class LimitedStream : Stream
{
    private Stream stream = null;
    private int limit = 0;
    private int totalBytesRead = 0;

    public LimitedStream(Stream stream, int limit)
    {
        this.stream = stream;
        this.limit = limit;
    }

    public override int Read(byte[] buffer, int offset, int count)
    {
        int bytesRead = this.stream.Read(buffer, offset, count);
        checked { this.totalBytesRead += bytesRead; }
        if (this.totalBytesRead > this.limit)
            throw new IOException("Limit exceeded");
        return bytesRead;
    }

    ...
}
```

Now, simply wrap the stream returned from WebResponse.GetResponseStream in a LimitedStream and return the LimitedStream from the GetEntity method (see **Figure 4**).

Figure 4 Using LimitedStream in GetEntity

```
private const int TIMEOUT = 10000; // 10 seconds
private const int MAX_RESPONSE_SIZE = 1024 * 1024; // 1 MB

public override object GetEntity(Uri absoluteUri, string role, Type
ofObjectToReturn)
{
    System.Net.WebRequest request = WebRequest.Create(absoluteUri);
    request.Timeout = TIMEOUT;

    System.Net.WebResponse response = request.GetResponse();
    if (response == null)
        throw new XmlException("Could not resolve external entity");

    Stream responseStream = response.GetResponseStream();
    if (responseStream == null)
        throw new XmlException("Could not resolve external entity");
    responseStream.ReadTimeout = TIMEOUT;

    return new LimitedStream(responseStream, MAX_RESPONSE_SIZE);
}
```

Finally, add one more defense-in-depth measure by blocking entity resolution of URIs that resolve to the local host (see **Figure 5**).This includes URIs starting with https://localhost, https://127.0.0.1 , and file:// URIs. Note that this also prevents a very nasty information disclosure vulnerability in which attackers can craft entities pointing to file://resources, the contents of which are then duly retrieved and written into the XML document by the parser.

Figure 5 Blocking Local Host Entity Resolution

```
public override object GetEntity(Uri absoluteUri, string role,
    Type ofObjectToReturn)
{
    if (absoluteUri.IsLoopback)
        return null;
    ...
}
```

Now that you've defined a more secure XmlResolver, you need to apply it to XmlReader. Explicitly instantiate an XmlReaderSettings object, set the XmlResolver property to an instance of XmlSafeResolver, and then use the XmlReaderSettings when creating XmlReader, as shown here:

```
XmlReaderSettings settings = new XmlReaderSettings();
settings.XmlResolver = new XmlSafeResolver();
settings.ProhibitDtd = false;   // comment out if .NET 4.0 or later
settings.DtdProcessing = DtdProcessing.Parse;  // comment out if
                                               // .NET 3.5 or earlier
settings.MaxCharactersFromEntities = 1024;
XmlReader reader = XmlReader.Create(stream, settings);
```

Additional Considerations

It's important to note that in many of the System.Xml classes, if an XmlReader is not explicitly provided to an object or a method, then one is implicitly created for it in the framework code. This implicitly created XmlReader will not have any of the additional defenses specified in this article, and it will be vulnerable to attack. The very first code snippet in this article is a great example of this behavior:

```
void processXml(string xml)
{
    System.Xml.XmlDocument document = new XmlDocument();
    document.LoadXml(xml);
}
```

This code is completely vulnerable to all the attacks described in this article. To improve this code, explicitly create an XmlReader with appropriate settings (either disable inline DTD parsing or specify a safer resolver class) and use the XmlDocument.Load(XmlReader) overload instead of XmlDocument.LoadXml or any of the other XmlDocument.Load overloads, as shown in **Figure 6**.

Figure 6 Applying Safer Entity Parsing Settings to XmlDocument

```
void processXml(string xml)
{
    MemoryStream stream =
        new MemoryStream(Encoding.Default.GetBytes(xml));
    XmlReaderSettings settings = new XmlReaderSettings();

    // allow entity parsing but do so more safely
    settings.ProhibitDtd = false;
    settings.MaxCharactersFromEntities = 1024;
    settings.XmlResolver = new XmlSafeResolver();

    XmlReader reader = XmlReader.Create(stream, settings);
    XmlDocument doc = new XmlDocument();
    doc.Load(reader);
}
```

XLinq is somewhat safer in its default settings; the XmlReader created by default for System.Xml.Linq.XDocument does allow DTD parsing, but it automatically sets MaxCharactersFromEntities to 10,000,000 and prohibits external entity resolution. If you are explicitly providing an XmlReader to XDocument, be sure to apply the defensive settings described earlier.

Wrapping Up

XML entity expansion is a powerful feature, but it can easily be abused by an attacker to deny service to your application. Be sure to follow the principle of attack surface reduction and disable entity expansion if you don't require its use. Otherwise, apply appropriate defenses to limit the maximum amount of time and memory your application can spend on it.

Bryan Sullivan is a security program manager for the Microsoft Security Development Lifecycle team, specializing in Web application and .NET security issues. He is the author of "AJAXSecurity" (Addison-Wesley, 2007).