

XML Threats

Much like HTML, in its earlier incarnations, XML was viewed as a simple and harmless markup language that grew in sophistication until “really bad things” could be done to computing systems using it. Note: XML does not harm computers; humans harm computers. This chapter discusses why XML-based platforms (including Web services and SOA) are being targeted, what types of attacks can be mounted, and how you can protect against them using DataPower.

The New Frontier

When we think of computer-based attacks, the Internet-based worms and viruses that have wreaked so much havoc in the past come to mind. These attacks have typically been mounted against Windows-based computers, primarily due to that particular operating system being “interesting,” as it runs on a large percentage of systems, and also because it has had plenty of vulnerabilities to exploit (particularly in earlier versions). However, these attacks seem to have died down in recent years, and this is for several reasons.

Like any technology, Windows has become more mature over time. Mature technologies are much more difficult to attack. Consider the changes in another product—IBM WebSphere Application Server (WAS). While the earliest versions had little security (in part due to the J2EE specification having little security itself), the newest versions are secure by default, with a sophisticated security model (again, partly due to the underlying Java EE specification now having a more robust security model).

Many products have been created as a means of defense against such attacks—such as anti-virus products that are constantly updated over the Internet for new attack profiles. Incoming email is scanned for viruses at the mail server before it is forwarded on to client inboxes. We can barely get through the work day without being bothered by our workstations downloading large operating system patches, wanting to install them and reboot our systems.

Attacking Windows-based systems is not the “badge of honor” it used to be. There are wizards and toolkits that can be downloaded to create viruses and worms, so this is not the proud technical achievement it once was. The sophisticated hackers have moved on to more interesting platforms and challenges.

The Technology Adoption Curve

The cycle of technology is repetitive. When new technologies appear on the radar, they are usually raw in terms of bugs, security, performance, and stability. For these reasons, businesses are reluctant to use them in production environments. As they mature, more confidence is gained and eventually they make it to the adoption phase and begin to appear in “interesting” production systems, even though they may not have a developed or well-tested security infrastructure.

By this time, even though the technology may have matured enough for businesses to have confidence in their viability, the staff may be inexperienced and make mistakes in the install or configuration that create vulnerabilities. This creates the “sweet spot” for attackers—interesting targets that have exploitable attack surfaces due to bugs or misconfiguration. As an example, consider Web services—a technology that has been around for quite some time. It seems that the WS-Security specification has taken forever to develop, having spent significant time in draft-12 and draft-13, and even version 1.0 had some problems. When specifications mature, it takes time for the products that are built based on them to appear with those new spec levels, and initially it is common to see bugs, as these specifications are typically complex and ambiguous in areas and, hence, hard to implement.

Although Web services have been around a long time, we are just now seeing common production implementations, now that a more fleshed out WS-Security 1.1 specification is found in the most popular Java EE runtime platforms. SOA and ESB-based systems are newer and have a lower position in the curve, but are also now quite common in production. All these XML-based platforms are now in that “sweet spot” for those wanting to compromise the systems that are running them.

Even newer technologies such as Web 2.0 and AJAX, which make for impressive demos, are frightening from a security perspective. As you might imagine, these are currently at the bottom of the technology adoption curve.

But, I Thought XML Was Our Friend!

Let’s go back to the early days of the World Wide Web, when all the amazing Internet technology we know today was in its infancy. HTML was born in order to display information delivered via the Internet over HTTP. For much of the early period of the Web, HTML was viewed as a harmless, innocuous text-based markup language. How could this possibly be harmful? In those friendlier days, it was inconceivable that anyone would or could use this fantastic technology to hurt others. As the need to display more interactive content grew and technicians (as they are wont to do) became more inventive, HTML grew from its simple beginnings to a complex and sophisticated technology. And, the more sophisticated and complex a technology is, the more

opportunity there is for building attacks and exploits. With the addition of scripting add-ons such as JavaScript, the potential to do “bad” things with HTML grew.

History has repeated itself with another markup language, XML, which began as a simple markup language to enable self-describing data. As the technology moved up the adoption curve, the specification grew and became more and more complex to accommodate more sophisticated use cases. XML now contains tags to cause parsers to do programmatic operations such as recursion. And where we have recursion, we know that we can do fun things like infinite looping, which are not much fun for computer systems! XML-related languages such as XPath and XSLT allow for even more mischief.

Dirty Little Secrets

It is primarily for these reasons that industry pundits have predicted an explosion of XML-based attacks, and in fact this has already begun to occur. It’s easy to dismiss the dire warnings about these types of attacks as hype simply meant to “sell products” because, after all, how many of these attacks has anyone heard of?

The truth of the matter is that by their nature, these types of attacks can conveniently be kept quiet. The earlier Internet attacks were mostly worms and viruses that targeted desktop computers through email. This means that the attacks affected everyone—computers at home, computers at work, computers at media outlets, computers in government—and so they are public. However, the “new frontier” attacks are specifically targeted at XML-based technologies—corporate Web services, ESB, and SOA runtime platforms. When a corporation is compromised by such an attack, it is usually a “targeted hit” on them alone, and thus, it is easier for them to keep quiet. They also have much incentive to do so—if a security compromise of a company in the competitive financial services or banking industry became public, it could cause them to rapidly lose the confidence (and business) of customers and shareholders.

Old Friends, Old Nemeses

These new types of attacks present some problems for the network topologies that have been stable and reliable for so long. Our old friends sitting out in the DMZ (the load balancers, proxies, and Web servers) that have always been the first line of defense and protected us can no longer help much against these attacks.

For example, Denial of Service (DoS) tactics have been a common way to mount attacks on networks. As these became popular, DMZ products were built to be smarter, for example, to monitor network traffic rates and take some action if the traffic profile fell into that suspected of being an attempted DoS. This would usually be something on the order of thousands of messages per second. The messages used for DoS often take the form of HTTP HEAD requests, which are small but deadly in the resources they consume in terms of connections used. The most effective and common way to deal with DoS has been simply to build a massive infrastructure, such as clusters consisting of many replicated, identical back-end runtimes. However, the XML-based attacks we discuss in this chapter go deeper into the application layer of the network stack than

simple HEAD messages, to leverage things specifically inside the application messages themselves. This is far outside the radar of traditional network products, which don't understand application message formats such as XML and SOAP.

You will see in this chapter that many of our old nemeses through the short history of computer attacks have come back to haunt us in new forms to take advantage of these more modern XML-based technologies. For example, the DoS attack we discussed in the previous paragraph can now be mounted using far fewer messages with even deadlier consequences. Each message, rather than an innocuous HTTP HEAD, is a small XML file that will by itself take down a single server in a cluster. So if you have a lot of computing power, say a cluster of 20 application servers, this chapter shows how an attacker would need only 20 messages to take down the entire system, rather than many thousands used in a traditional DoS attack.

There are other products designed specifically to help prevent XML-based attacks—but some of them are built on complex runtimes, such as Java Runtime Environment (JRE) or Java EE, which are not suitable for DMZ deployment due to their large attack surface. This means they must be deployed *behind* the DMZ, as shown in Figure 20-1, and your message traffic then incurs yet another hop before reaching its destination. Of course, this is also unfortunate as it would be optimal to detect and repel any attack in the DMZ, as that is its purpose. It is a frightening thought to allow any potentially malicious message into the private/trusted zone. DataPower can replace both the Web and XML Sentry servers and accomplish the goal of heading off threats in the DMZ.

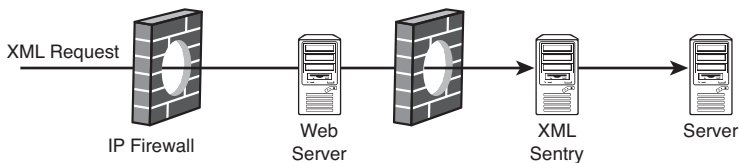


Figure 20-1 Excessive network hop due to the software-based XML security tool.

XML Threat Categories and Examples

We started this chapter with a lot of strong talk. Let's drill down into the specific details and show some examples.

Four Categories of XML Threats

Although there are a lot of blurred lines and gray areas, XML-based threats can generally be broken out into four major categories:

- **XML Denial of Service (xDOS)**—These attacks are meant to slow down or disable an XML-based system so that valid service requests are slowed or fail. This category can be broken down even further into single and multiple message XML-DoS.
- **Unauthorized Access**—These stealth attacks are designed to gain unauthorized access to a backend service or its data.

- **Data Integrity/Confidentiality**—These attacks strike at data integrity of XML-based responses, requests, or underlying data repositories.
- **System Compromise**—These attacks corrupt or attempt to gain control of backend systems.

Some of the threats that we discuss fit into more than one of these categories. In general, some of them are stealthy, “sneak” attacks where the attacker does not want to be detected, while others are quite obvious.

Single-Message Denial of Service Attacks

Single-message attacks can be mounted in several ways. As we discussed earlier, some of these attacks are so potent that a single message can wreak havoc. Let’s look at a few examples.

Jumbo Payload Attacks

XML, by its nature, is resource-intensive to process. It was designed to make data easier to understand for humans and computers. Prior to XML and other self-describing data formats, text or binary data would typically be formatted in large blocks and one would need a separate template, or roadmap, that would outline where the different fields and records began and ended. Those more experienced among our readers may remember poring over long sheets of green-bar paper, pencil in hand, delineating each field in a hex dump to look for problems. “Let’s see—I believe the street address should begin here, in position forty-two, but it looks like we’re off by one....” With XML, this is no longer necessary, as the file itself contains the metadata to delineate and describe each field in the document via its tag name. This approach has many advantages; among them is the ease with which humans and computers can find, validate, and work with the data in the message.

However, this convenience comes at a cost. XML documents are read into memory in a process called *parsing*, which also serves to create a document object model of addressable nodes. The larger or more complex the document, the more intensive this parsing process is on the host system. Parsing is not overly CPU-intensive—the real hit comes in memory utilization. The in-memory representation of an XML document is normally expanded due to additional “leaves” in the parsed tree representation being created as part of the process, and as a result of processing instructions that might be in the XML document and can lead to great expansion during the parsing process. (We discuss this later in this chapter.)

All this is bad enough as far as the effects on a computer system. Now imagine the consequences of parsing huge XML files. In fact, some application XML message schemas are designed with full knowledge that the resulting XML files will be enormous—we’ve seen examples measured in *gigabytes*. We call these “self-inflicted wounds!”

Hackers know this, too, which is why one of the simplest forms of attack is to send large XML payloads. There are many forms of jumbo payload attacks, from the XML markup itself to large SOAP attachments. You might say, “Well, the payload has to be valid—this is what we have schema validation for,” but we would counter that the XML input must be parsed *before* it can be validated in most cases, and many of these are parser-based attacks. As for the many systems we’ve seen where schema validation is not done for performance or other reasons, well let’s not go there!

Recursion Attacks

As we mentioned earlier, XML has grown in complexity and sophistication, moving from a simple, static markup language to a dynamic, full-blown programming model (as implemented in XSLT). One of the programming constructs supported is recursion. And where we can have recursion, we can have infinite or long-lived looping constructs, which are ideal for consuming processing power or resources in the form of memory and file system space. As an example, review Listing 20-1, which is a small XML file. (The listing has been abbreviated for conciseness.)

Listing 20-1 Billion Laughs Recursion Attack

```
<?xml version="1.0"?>
<!DOCTYPE billion [
<!ELEMENT billion (#PCDATA)>
<!ENTITY laugh0 "ha!">
<!ENTITY laugh1 "&laugh0;&laugh0;">
<!ENTITY laugh2 "&laugh1;&laugh1;">
...
<!ENTITY laugh127 "&laugh126;&laugh126;">
]>
<billion>&laugh127;</billion>
```

This is called the “Billion Laughs” attack—without going too far into the nuances of XML trickery, you can see that this file has a series of ENTITY entries, each of which references and expands to the ones above it. So the file grows exponentially in memory when it is parsed, consumes CPU cycles, and mushrooms in size to eat up the memory space of its host computer, particularly Java heaps. The resulting XML file (again abbreviated) is shown in Listing 20-2.

Listing 20-2 Billion Laughs Output

```
<billion>
ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!
ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!
ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!
ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!
ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!
ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!ha!
.....
</billion>
```

For an easy to reproduce demonstration, we cut down the Billion Laughs XML in size (to just laugh12) and loaded it with a commonly used Web browser (up to date with all patches) on a fast dual-processor machine with 3GB of memory. Figure 20-2 shows the result—immediately one of the two CPUs was consumed, the system was paging and memory usage quickly spiraled upward. (We killed the process at half a gigabyte so that work on this chapter did not become lost!) To be fair, this scenario crashed every browser we tried, not just the one shown.

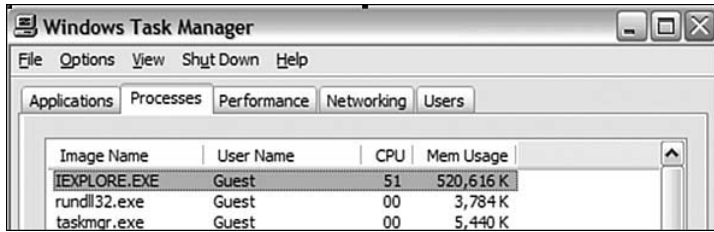


Figure 20-2 A Billion Laughs browser attack.

Mega-* Attacks

The XML specification does not mandate things such as limits on the length of tags (for example element tag names), the number of namespaces in a document, or the maximum number of nested levels in a nodeset. So, we can create well-formed XML documents with mega-tag-names, mega-nesting, mega-namespaces, and so on. These can be used to cause a number of problems. Long element tag names can surprise parsers by exceeding their expectations, causing buffer over-runs in some systems and in others raising system error conditions that may cause processing failures. Extreme levels of nesting can be accomplished by using the recursive techniques we just described. Adding mega-namespaces is trivial, something a person with a rudimentary understanding of XML might understand and attempt (read: young hackers who have had a few intro computer classes). Listing 20-3 shows a SOAP message with 9,999 namespace declarations. (You probably don't need anywhere near that many.)

Listing 20-3 A SOAP Message with Mega-Namespaces

```
<S:Envelope xmlns:S='http://schemas.xmlsoap.org/'>
<S:Body xmlns='http://example.com/'>
<X xmlns:X1='http://www.example.com/x1'
  xmlns:X2='http://www.example.com/x2'
  .....
  xmlns:X9998='http://www.example.com/x9998'
  xmlns:X9999='http://www.example.com/x9999'>
</X>
</S:Body>
```

The parser attempts to read and make some sense of these declarations, which causes an extreme amount of overhead. In testing this attack, typically as soon as the input file hits the run-time parser, CPU usage shoots directly to 100 percent utilization and stays there, as shown in Figure 20-3. The system becomes unresponsive to legitimate messages and does not respond to system shutdown commands, leaving the only options as doing a forced power switch shutdown or riding it out. These are not good options—a power off would likely result in a mess of broken transactions to repair, and riding it out means a denial of service to good messages and possibly a full system outage.

Remember, if you have a large cluster of perhaps 20 nodes, an attacker needs only 20 small messages to bring down every server (assuming they are clever enough to send from clients with different IP addresses to avoid stickiness to one particular server). DoS policies set on traditional network equipment would not even blink at 20 messages.

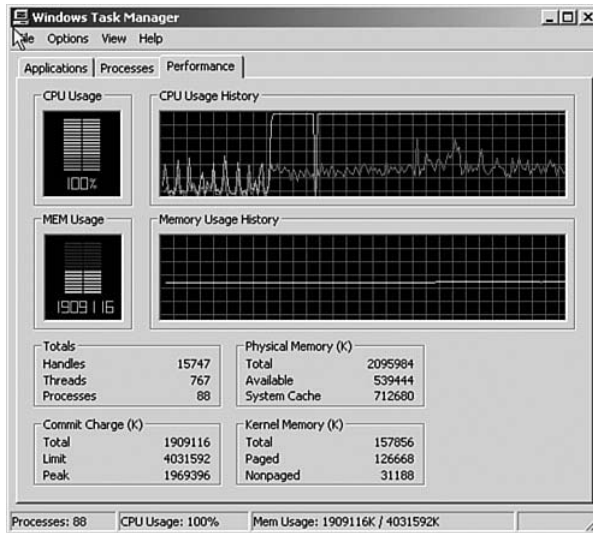


Figure 20-3 CPU consumption during a mega-namespace attack.

Coercive Parsing Attacks

Coercive parsing attacks entail using the sophistication of the XML language to write files that will give the parser fits. Examples might be circular references within or outside of the host document. Legacy systems that implement XML but do not have modern parsers, or have low processing capabilities, are particularly susceptible to these attacks. The Billion Laughs attack could be considered an example of coercive parsing.

Public Key DoS Attacks

In Chapter 18, “DataPower and SSL,” and Chapter 19, “Web Services Security,” one fundamental point was made clear: Security tasks involving asymmetric keys are expensive computational operations. They consume plenty of resources when working as intended. So what better way to mount an attack than to leverage already powerful and resource-intensive processes like these? Public Key DoS attacks are mounted by sending messages with signatures or encrypted data that use very strong cipher strengths and long key sizes. If certificate chain checking is employed, or if Certificate Revocation Lists (CRLs) are being used, the process can take even longer.

Multiple-Message Denial of Service Attacks

All the attacks in the previous section were contained within a single XML document, and as you have seen, can cause considerable grief for computer systems. Multiple-message attacks are mounted by sending more than a single message to a particular system. Let's look at a few of those.

XML Flood

We've seen the damage a single XML document can cause. Consider the impact if we decide to send a flood of them! That's what the XML Flood attacks are all about. Pick any one of the single-message attacks, or a healthy mix, and set up a script to bombard a system, and you can imagine the result.

Even small, benign messages can be sent in rapid succession by using the type of load-producing tools intended for performance testing to wreak havoc on systems, simply due to the overhead of parsing and error handling, in addition to the other DoS symptoms such as socket/connection saturation.

Resource Hijack

We often think of hackers, or those seeking to compromise our systems, as bad people dressed in dark, shabby clothing and hiding out in remote corners of the world. However, according to studies many attacks originate on the *inside* of corporations, not the outside! Resource hijack attacks involve some knowledge of the applications and their transactional properties.

For example, database deadlock (also known as “deadly embrace”) is a problem that has been around for a long time. Despite all the fancy technology out front and in the middle of our topologies, in most cases, it is still ultimately a database that the information is stored in. If someone knows the proper transaction sequence from a client perspective—perhaps a series of Web services calls—to cause resource contention and deadlocks on the backends of our systems, the result can be an outage.

Unauthorized Access Attacks

Unauthorized access attacks are just what they sound like—attempts to gain access to information that one would not normally be permitted to access. There are several known forms of this type of attack, as discussed in the following sections.

Dictionary Attacks

We mentioned earlier in this chapter about our “old friends” coming back to see us, dressed in new clothes and hoping for acceptance this time around. Dictionary attacks have been around for as long as there have been Web login forms. The concept is that people are basically lazy about passwords, because “good” passwords are hard to remember. A great password would be something like ‘@#GWse5@%&8!_4’, but it would be hard to remember, so instead in many cases we might go with the ever so tricky “passw0rd” or the name of our first born child, and then write it on a yellow sticky note and attach that to the underside of our keyboard for layered security.

Attackers know that there are a core set of passwords that many people use. Dictionary attacks are mounted by using an automated script and database of common passwords to “keep on trying” until one password succeeds. In the case of Web login forms, a script would continually enter passwords from the common password database (assuming the userid is known, or else it might be a combination of user ID and password dictionaries) until one works. However, at some point we got smart—and the new “best practice” for login forms became the “three strikes rule”—which means that after three straight bad passwords, you are no longer allowed to log in until calling the help desk (or some ‘timeout’ period may be enforced).

What about new ways of logging in? User IDs and passwords are found in many different places now—not just login forms. In Chapter 19 we discussed the WS-Security UsernameToken profile, which contains a userid and password. Is anyone enforcing “three strikes” on those? Most likely they are not.

Falsified Message and Replay Attacks

Falsified message and replay attacks depend on intercepting legitimate messages and then altering them. Intercepting messages is also referred to as “man-in-the-middle” and can be thwarted by using the network security techniques we’ve described throughout this book, such as mutual SSL tunnels and network interface isolation.

An example scenario might be an attack where a SOAP message that updates payroll information is intercepted and studied, and used to submit additional, similar messages later, perhaps modified to provide a “bump” in salary for a particular employee.

The message may have been sent by someone with privileges to make such changes, such as the payroll administrator. But if someone else on the network (perhaps an operator who feels she is not properly compensated) intercepts and modifies that message, it will still be sent with the payroll administrator’s identity if that credential is embedded in the message. This highlights the importance of encrypting all network traffic, and setting good timeout values on credentials that are being created for the message payload—they should always have a limited lifetime in order to minimize the time available for such manipulations.

TIP — INTERNAL SSL

Many network topologies show that SSL is used only to protect connections to and from the outside world, and then is not used for internal network connections. This is just the type of configuration that allows man-in-the-middle scenarios. Keep in mind that if a Data-Power appliance in the DMZ has an SSL connection to client-facing traffic and a non-SSL connection from the DMZ to the private internal network on the same Ethernet interface, you are still allowing unencrypted sensitive data in the DMZ, which can be sniffed and compromised. This is why using separate interfaces for inbound (client-facing) and back-end (to the trusted zone) provides better security through network isolation.

Even for systems completely inside the internal network, without SSL you are opening yourself up to sniffing and information compromise by your employees or contractors, which in some cases may put you at risk of lawsuits or in violation of laws meant to protect others’ privacy.

Data Integrity and Confidentiality Attacks

Data integrity and confidentiality attacks can take many forms. They can exploit weaknesses in the access control mechanism that permits the attacker to make unauthorized calls to the Web service to alter or view data on the backend, or by examining all or part of the content of a message to breach the privacy of that message's content. This is called *message snooping*.

Message Snooping Attacks

Snoop attacks can happen to messages being transmitted in the clear or transmitted encrypted but stored in the clear, or by decrypting messages using stolen key or crypto material. This is why we have filled this book with encouragement to use crypto for message integrity and privacy, and above all to protect your private keys!

In Chapter 4, “Advanced DataPower Networking,” we covered the basics of the Ethernet protocol, on which almost all networks are based. We discussed how messages bounce around the network until they find their intended target. There are many tools available that are capable of intercepting and displaying messages from the network. These are the type of tools that are employed not only for network troubleshooting, but also sometimes for bad intent. There are many tales of salary information, sensitive email, love letters, embarrassing photographs, and other types of messages being intercepted within a workplace and leading to interoffice mayhem, or worse.

TIP—NEVER SNIFF THE NETWORK WITHOUT PERMISSION!

Because these network sniffing tools can compromise privacy, their use is a very sensitive issue for corporate IT departments. If you ever feel the need to use one (for legitimate purposes we hope), it is best to ask for permission in writing from the network or IT security team. Network admins and security sentries can actually “sense” when these sniffers are activated over the network. Don't be surprised to get a visit rather quickly at your workstation should you do this—they have ways of tracking you down!

SQL Injection Attacks

SQL injection attacks, of course, attack databases. These are yet another new twist on an old form of attack. In past days, someone figured out that the data that is entered in computer entry forms is, in most cases, inserted directly into a SQL command by the backend application.

For example, if a manager has a form that lets her pull contact information for any of her direct reports, but not others in the company, she would log in and the application might show a form with a drop-down list of only her direct reports from which to choose, based on her login ID. She could choose an employee ID and check off the pieces of info that she wants—address, mobile phone number, and so on. For example, if she selects the ID ‘j_smith’ from the drop-down and checks off certain fields in the form to get Jerry Smith's business contact info, behind the scenes an application might take what she enters and use it to build a SQL statement with code, such as that shown in Listing 20-4.

Listing 20-4 Typical SQL Statement Built Using Programming Code

```
SQLStmt= "SELECT " + fieldString + " FROM corp_employees WHERE username =  
'" + username + "';"
```

Now imagine the effect on that programming code, and the resulting SQL command string, if one were to get access to the data entered in the form and change it before it gets to the backend application. The value 'j_smith' could be substituted with the following malicious values;

- '' would result in all employees' contact info being returned—a breach of privacy because this manager was only allowed to see his own direct reports. "OR 1=1" could also be appended to j_smith for the same result.

'j_smith');DROP TABLE corp_employeesj--' uses the semicolon command delimiter to append another command on the end, this one would delete the table!

- j_smith');INSERT INTO db_admins (user_id, passwd, is_admin) VALUES (badguy01, mypwd, true);--' would stealthily insert an entry into the database administrator table for the hacker, and from there all bets are off because they have full control of the database.

If the column names to be returned by the SQL query were also changed, for example to compromise the value of fieldString in Listing 20-4, we could get a lot more information on the employees than just their addresses. For example, if this were changed to "*" then all columns would be returned, possibly exposing salary rates, drivers license numbers, and other sensitive information. Using a value of 1/0 for this field results in a divide by zero, something that is never fun for computer systems to attempt!

Just as we have learned in the past how to protect against attacks like DoS, we have also learned how to protect against these form-based attacks. The solution is to check all field entries on form posts for any suspicious looking strings, such as SQL commands, JavaScript, or system commands. However, in today's world, much of the data heading to backend databases is contained in SOAP messages, and it is unlikely they are being subjected to this type of sanitizing. Not that there is a whole lot funny about cyber attacks, but Figure 20-4 from the wonderful IT comic xkcd (<http://www.xkcd.com/>) puts SQL injection into a humorous light.



Figure 20-4 A little SQL injection humor.

In fact, there is a documented case of someone using a custom license plate on a vehicle to try to gain revenge against law enforcement using license-plate scanning technology to catch vehicle infractions—the custom license plate reads OR 1=1;-- (see <http://www.areino.com/hackeando/> for more information).

XPath/XQuery Injection

In Chapter 22, “Introduction to DataPower Development,” we introduce you to XPath. It is a language for addressing and querying parts of an XML document, as opposed to databases. Similar to SQL, injection attacks can also be applied to XPath statements. With databases, some SQL injection attacks may be denied simply due to the user not having access to the particular columns being requested. Access is often controlled by ACLs on the database itself. However, there is no such method of authorizing for access to specific areas of an XML document with XPath, so it is all the more dangerous.

Follow-on technologies to XPath such as using XQuery with XACML do provide some relief in this area, but again only if these new and somewhat complex tools are implemented in the products you use, understood by your developers, and configured correctly.

WSDL Enumeration

You learned in Chapter 10, “Web Service Proxy,” that a WSDL file is a virtual handbook on the details of a Web service application. The available operations are listed, along with the host names, ports, and URIs. This is very convenient for publishing to registries and for building clients, intermediaries, and proxies, such as with DataPower. However, we have talked a great deal in this book about the importance of not exposing “too much” in the way of implementation details so that the information cannot be used to mount an attack. The WSDL in particular poses a problem in this light.

Often, WSDL files are generated blindly from development or directory tools. They will examine a Web services application and spit out everything that is found therein to the resulting WSDL. Consider the normal case, where programmers have not only implemented the services they were tasked with, but along the way built in other services to aid their debugging, or maybe aborted attempts or older versions of the operations that are still laying around in the code. Sometimes programmers will build in their own “back door” services to circumvent things like security for their own testing.

If a WSDL is built with these services exposed, they can be discovered by WSDL enumeration. There are several ways that inquiring minds can get access to the WSDL. If it is published in a public directory, of course the job is easy. Most hackers know that you can simply append `?wsdl` to the end of any Web services URL, and as part of the specification, the backend WSDL is returned to the client. Now, wasn’t that easy!

The WSDL file should be treated like application code, and as such subject to a thorough review by multiple parties, as part of a stringent process to ensure security. When publishing services to directories, care should be taken to publish only those that are intended to be public.

Schema Substitution

The XML schema definition file is the template that XML input and output files must adhere to. It lays out the acceptable structure of the documents, which elements are required versus optional, and what order things should be in. It is meant to be used to enforce “good” data coming into the system and head off “bad” data at the pass by easily detecting it.

XML files can identify the schema that they should conform to by including a link inside of themselves. Listing 20-5 shows a simple XML document containing a reference to its schema on a remote server, and what that schema file might look like. You can see that the types are tightly defined. Seeing loose definitions such as `xs:any` for field types should be a red flag, as these will allow any type of content, including perhaps our friend the Billion Laughs!

Listing 20-5 An XML Document with Schema Reference and Its Schema File

```
<product xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="http://myserver/schemas/product.xsd">
  <prod_id>001234</prod_id>
  <price>42.36</price>
</product>

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="product" type="product"/>
  <xs:complexType name="product">
    <xs:sequence>
      <xs:element name="prod_id" type="xs:string"/>
      <xs:element name="price" type="xs:decimal"/>
    </xs:sequence>
  </xs:complexType>
</xs:schema>
```

The danger of this approach is in the server hosting that schema possibly becoming compromised, and the schema being replaced with one that allows malicious content. When you are depending on remote servers, you are extending your trust domain, and must do so judiciously, as you are then only as secure as those other servers. Also, in a case where the remote server is down (perhaps due to an attack), you risk being brought down due to the inaccessibility of the files. Remote servers should be highly available and redundant if they are critical.

Routing Detours

Routing detours involve using the SOAP routing header to access internal Web services that perhaps were not meant to be accessed. With technologies such as WS-Addressing, this is even easier if one can get access to and modify or replay the client message. It would then be possible to route messages to an alternate server, perhaps one owned by the hacker. These could even be copies of messages so that it may never become obvious that something bad is happening.

System Compromise Attacks

While some of the previously discussed attacks might be stealthy in nature, there's nothing subtle about this category. The general intention of a system attack is to raise hell in a profound way. Let's look at some examples.

Malicious Include Attacks

Like most programming dialects, XML allows for directives to include other files within a file. So why not take advantage of this by inserting a directive to include a file such as `/etc/passwd`? Before you assume the user will not have privileges to access this type of file, keep in mind that many backend server processes that serve up XML responses are running with root privileges.

Memory Space Breach

The memory space breach attack has been around for a long time as well. Like the injection attacks, it relies on inserting some data into an entry form. This time, the goal is to use very large data such that the memory space of the system might be exceeded and the data might "bleed" over into the area of memory that contains processing instructions. And if the data that is being entered contains processing instructions, those instructions might be executed. Memory space breaches are accomplished by causing stack overflows, buffer overruns, and heap errors. While this is more difficult in Java-based systems, it can still be done.

XML Encapsulation

CDATA is an XML construct (short for character data) to include content that is "not XML" in an XML file. Using this tag tells the parser that this is character data, and it should not be parsed. This can be used to include nonlegal characters in XML, including system commands and script. Listing 20-6 shows a CDATA section that you do not want executing on your system!

Listing 20-6 CDATA Being Used for Bad Intent

```
<![CDATA[  
badCmd = new ActiveXObject( "WScript.Shell" );  
badCmd.Run( "%systemroot%\SYSTEM32\CMD.EXE /C DEL C:\\*.*" );
```

XML Viruses

Sound familiar? We've talked a lot about how many of these "XML Threats" are new twists on traditional attacks. Most people who use computers know what viruses are, and how they get them. The common delivery vehicle is through an email attachment. We learned our lesson here as well—these days our email is scanned for any virus-laden attachments at the mail server before it is permitted to be forwarded to the email client inbox on our workstations. The virus scanners are updated frequently (as our desktops are!) with the latest known virus signatures, as new viruses are appearing all the time.

What about SOAP attachments? Is anyone scanning those for viruses? These are even more destructive as they are opened on the server, rather than an email client at a desktop. SOAP attachments are just as capable of carrying viruses as email attachments, yet they are rarely checked.

Threat Protection with DataPower

Now that we've sufficiently scared you from ever using computers again, let's get to the good news. DataPower appliances are an effective way to protect against these types of threats. This is primarily due to the end-to-end XML security focus of the products, as well as having the horsepower to pull off the intense introspection and interrogation of messages for suspicious profiles without a major hit to performance. Let's walk through some of the configurations that can save you.

Characterizing Traffic

Several of the attacks we discussed were based on sending badly formed data. Most traditional DoS attacks send huge quantities of HTTP HEAD requests, which contain no body. One simple first step toward protecting your systems is to characterize the type of messages you are expecting. In Chapter 8, "XML Firewall," we showed the Request Type and Response Type fields. These also appear in the Multi-Protocol Gateway and Web Service Proxy. Figure 20-5 shows these drop-downs and the possible choices.

If your service is intended to receive only SOAP messages, even though the other choices will allow SOAP, by being more specific you are enabling the device to help protect you by telling it to reject anything that is not SOAP (and even badly formed SOAP). Those messages could be malicious. Some might get lazy here and just specify XML, and that will work, as SOAP is an XML vocabulary. But you are then subjecting yourself to any XML-based attacks (such as the Billion Laughs attack) that aren't based on SOAP and would otherwise be rejected.

The screenshot displays the configuration interface for a DataPower appliance, divided into two main sections: Back End and Front End.

Back End Configuration:

- Server Address:** myserver.myco.com *
- Server Port:** 80 *
- SSL Client Crypto Profile:** (none) [dropdown] [plus] [dots]
- Response Type:** SOAP [dropdown]
- Response Attachments:** Strip [dropdown]

Front End Configuration:

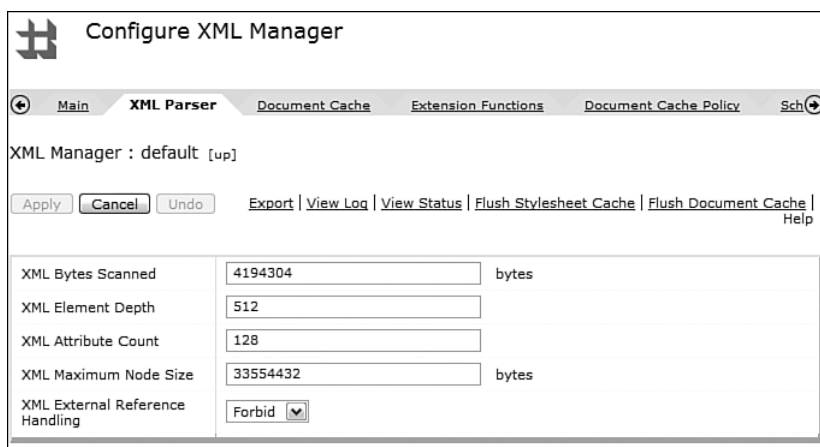
- Device Address:** UserInterface [button: Select Alias] *
- Device Port:** 12000 *
- SSL Server Crypto Profile:** (none) [dropdown] [plus] [dots]
- Request Type:** SOAP [dropdown]
 - SOAP
 - Non-XML
 - SOAP
 - Pass-Thru
 - XML
 - Strip
- Attachments:** [dropdown]

Figure 20-5 Characterizing front and back side traffic.

XML Manager Protections

A deeper level of message characterization can be found in the XML Manager. Figure 20-6 shows the XML Manager XML Parser tab. You should have a good idea of the characteristics for your own request and response documents from your testing procedures. (You do have those, right?) While the defaults here are reasonable, there is no way for the fine engineers at DataPower to know what is “right” for everyone else’s applications.

If you know that the maximum nesting depth of your messages is 14 levels, you can tune these settings much more efficiently by lowering the default of 512. This particular setting is useful for preventing recursion attacks that attempt to deeply nest the document.



The screenshot shows the 'Configure XML Manager' dialog box with the 'XML Parser' tab selected. The dialog has a title bar with a DataPower logo and the text 'Configure XML Manager'. Below the title bar is a tabbed interface with tabs for 'Main', 'XML Parser', 'Document Cache', 'Extension Functions', 'Document Cache Policy', and 'Sch'. The 'XML Parser' tab is active. Below the tabs, the text 'XML Manager : default [up]' is displayed. A row of buttons includes 'Apply', 'Cancel', 'Undo', 'Export', 'View Log', 'View Status', 'Flush Stylesheet Cache', 'Flush Document Cache', and 'Help'. The main area contains five settings, each with a text input field and a unit label: 'XML Bytes Scanned' (4194304 bytes), 'XML Element Depth' (512), 'XML Attribute Count' (128), 'XML Maximum Node Size' (33554432 bytes), and 'XML External Reference Handling' (Forbid, shown in a dropdown menu).

Setting	Value	Unit
XML Bytes Scanned	4194304	bytes
XML Element Depth	512	
XML Attribute Count	128	
XML Maximum Node Size	33554432	bytes
XML External Reference Handling	Forbid	

Figure 20-6 XML Manager XML Parser tab for XML characterization.

The other fields shown can (and should) be similarly tuned. The Bytes Scanned, Attribute Count, and Maximum Node Size fields are self-descriptive and can prevent mega and jumbo payload attacks. In particular, the Attribute Count setting would prevent the attack we demonstrated earlier, which depended on a large number of namespace declarations.

One more item of note before we leave this topic, is the XML External Reference Handling field. This can be used to tell DataPower to stop parsing if it encounters an external reference (either to an XSD, entity, or DTD) to ignore these and replace them with empty strings, or to allow them. The most secure setting (Forbid) is the default. This is here specifically to prevent substitution attacks.

Network/Protocol Protection

Figure 20-7 shows the Advanced Networking tab from the XML Firewall (similar options appear on all services or FSHs). We discussed how DoS attacks typically use HTTP HEAD or GET messages as they are lightweight. The Disallow GET (and HEAD) configuration will cause those to be rejected. This is on by default—if your applications use HEAD or GET, turn this off to allow

those message types. Disabling GET on the FSH for a Web Service Proxy also prevents the WSDL from being returned to the client with the ?wsdl technique that we discussed in the section on WSDL Enumeration attacks.

The ability to set up ACLs is also seen in Figure 20-7. This allows you to designate only certain other computers that are allowed to connect to your services. While this might not be useful for workstations due to the predominant use of DHCP (although you can specify ranges and netmasks), it can be useful for server-to-server configurations inside the trusted, internal network.

Advanced Networking

HTTP Timeout
120

TCP Timeout for Persistent HTTP
180

Enable Persistent Connections
☒ on ☐ off

Enable Compression
☒ on ☐ off

Suppress "Transformation Applied" Warnings
☒ on ☐ off

Rewrite "Host:" Header
☒ on ☐ off

Disallow GET (and HEAD)
☒ on ☐ off

Advanced Crypto

Firewall Credentials
(none) + ...

Header for Client Address
X-Client-IP

Default Param Namespace
http://www.datapower.com/param/config

Query Param Namespace
http://www.datapower.com/param/query

SOAP Schema URL
store:///schemas/soap-envelope.x

Access Control List
(none) + ...

Service Priority
Normal

Figure 20-7 The Advanced Networking configuration pane.

TIP — DON'T DEPEND ON IP-BASED PROTECTION

Keep in mind that IP addresses can be spoofed, so any security configuration that depends on them should be used as a front line of defense, with stronger measures backing it up. Security in layers is a good thing!

The XML Threat Protection Tab

The XML Threat Protection tab is the centerpiece for configuring against attacks. It is found on all the primary services, and there are sections within it for many of the major attacks that we have discussed. Let's have a look at those now.

Single-Message DoS Protection

Looking at this section, as shown in Figure 20-8, we can see that it is similar to the XML Manager page shown in Figure 20-6. We see many of the same fields here, and as you can imagine, there would be a conflict if the settings were defined differently in both places. You can use the Override XML Manager parser limits radio button to specify that you want the settings here, on the service, to take precedence over the XML Manager. If you have several services using the same profile settings, it may be more efficient to configure them in an XML Manager and share that between the services. One addition here is the Max. Message Size, which is for the total message size, including headers and other metadata. It defaults to zero, which means no limit is enforced (in lieu of the more specific XML settings in most cases). Another field allows setting limits on the size of any incoming attachments, again to protect against jumbo payload attacks.

Another interesting field in this section is the Recursive Entity Protection, which would protect against recursion attacks. Notice that this field is disabled for changes, meaning it is always on—you don't get a choice for this one!

Configure XML Firewall

General Advanced Stylesheet Params Headers Monitors **XML Threat Protection**

Apply Cancel

Miscellaneous XML Threat Protection Configuration

This page lets you configure the device for protection against the following XML threats:

- Single Message XML Denial of Service (XDoS) Protection
- Multiple Message XML Denial of Service (MMXDoS) Protection
- Message Tampering Protection
- SQL Injection Protection
- Protocol Threat Protection
- XML Virus (X-Virus) Protection
- Dictionary Attack Protection

Single Message XML Denial of Service (XDoS) Protection

Max. Message Size

Override XML Manager parser limits ☒ on ☐ off

Max. XML Attribute Count *

Max. XML Bytes Scanned *

Max. XML Element Depth *

Max. XML Node Size *

Attachment Byte Count Limit *

XML External Reference Handling ▼

Recursive Entity Protection ☒ on ☐ off

Figure 20-8 XML Threat Protection Tab and Single-Message DoS.

Testing Single-Message DoS on DataPower

Now that we've shown how DataPower can protect against single-message attacks, let's demonstrate what happens when we throw two specific examples we gave earlier at it. For this test, we configured a simple loopback XML Firewall with all defaults. Figure 20-9 shows the result of the Billion Laughs attack. It is apparent here that the single-message default for maximum message size was exceeded and caught this when the input message mushroomed beyond the limit of 4,194,304 bytes.

```
xmlfirewall (SimpleLoopbackFW): No match from processing policy 'SimpleLoopbackFW' for code '0x00030003'
xmlfirewall (SimpleLoopbackFW): XML parser limits exceeded
xmlfirewall (SimpleLoopbackFW): Generated error on URL 'http://192.168.1.130:2048/': <?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"><env:Body><env:Fault><faultcode>env:Client</faultcode><faultstring>Malforme
content (from client)</faultstring></env:Fault></env:Body></env:Envelope>
xmlfirewall (SimpleLoopbackFW): XML parser limits exceeded
xmlfirewall (SimpleLoopbackFW): rule (SimpleLoopbackFW_request): implied action Parse input as XML failed: document size limit of 4194304
bytes exceeded, aborting
xmlfirewall (SimpleLoopbackFW): document size limit of 4194304 bytes exceeded, aborting
xmlfirewall (SimpleLoopbackFW): Parsing document 'http://192.168.1.130:2048/'
xmlfirewall (SimpleLoopbackFW): rule (SimpleLoopbackFW_request): selected via match 'SimpleLoopbackFW' from processing policy
'SimpleLoopbackFW'
xmlfirewall (SimpleLoopbackFW): New transaction(conn use=1): POST http://192.168.1.130:2048/ from 192.168.1.107
```

Figure 20-9 Log entries from Billion Laughs message submitted to the device.

Figure 20-10 shows the log entries for a mega-namespaces attack. This time, the maximum attribute limit of 128 caught the offending message.

```
xmlfirewall (SimpleLoopbackFW): No match from processing policy 'SimpleLoopbackFW' for code '0x00030003'
xmlfirewall (SimpleLoopbackFW): XML parser limits exceeded
xmlfirewall (SimpleLoopbackFW): Generated error on URL 'http://192.168.1.130:2048/': <?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"><env:Body><env:Fault><faultcode>env:Client</faultcode><faultstring>Malforme
content (from client)</faultstring></env:Fault></env:Body></env:Envelope>
xmlfirewall (SimpleLoopbackFW): XML parser limits exceeded
xmlfirewall (SimpleLoopbackFW): rule (SimpleLoopbackFW_request): implied action Parse input as XML failed: attribute limit of 128 per
element exceeded, aborting at offset 6190 of http://192.168.1.130:2048/
xmlfirewall (SimpleLoopbackFW): attribute limit of 128 per element exceeded, aborting at offset 6190 of http://192.168.1.130:2048/
xmlfirewall (SimpleLoopbackFW): Parsing document 'http://192.168.1.130:2048/'
xmlfirewall (SimpleLoopbackFW): rule (SimpleLoopbackFW_request): selected via match 'SimpleLoopbackFW' from processing policy
'SimpleLoopbackFW'
```

Figure 20-10 Log entries from mega-namespaces message submitted to the device.

Multiple-Message DoS Protection

The Multiple-Message DoS Protection, as shown in Figure 20-11, is turned off by default. It is used to set up policies for rates and durations of messages coming into the device.

As the figure demonstrates, we have enabled this and provided some sample values for discussion. Starting from the top, our sample configuration will not allow the maximum duration for any request to exceed 5 seconds.

Next, we are configuring so that more than ten messages per second from any one host (client) will cause an over-threshold condition and subsequent messages within that threshold

period (one second) will be denied. Similarly, we are allowing for a maximum of 1,000 messages per second to this service overall (from all clients cumulatively). If either the host or firewall thresholds are exceeded, we have configured a block interval of half-second where no traffic will be permitted. If any of these thresholds are exceeded, a message will be logged at the specified severity.

These settings can be used in conjunction with performance test results from your backend systems to ensure that they are never over-run. For example, if performance testing on your backend application servers show that the JVM craters when transaction rates hit 1,000 per second, you can set a policy here to ensure that the rate of traffic to that server is never exceeded. However, the SLM policies, where available (WSP and MPGW), are more sophisticated ways to do this, and are also shareable and enforceable between peer devices.

Multiple Message XML Denial of Service (MMXDoS) Protection

Enabling MMXDoS will create duration and count monitors and attach them to this firewall.

Enable MMXDoS Protection

☒ on ☐ off

Max. Duration for a Request

msec *

Interval for Measuring Request Rate from Host

msec *

Max. Request Rate from Host

messages/interval *

Interval for Measuring Request Rate for Firewall

msec *

Max. Request Rate for Firewall

messages/interval *

Block Interval

msec *

Log Level

error

 *

Figure 20-11 Multiple-Message DoS Protection.

Protocol Threat Protection

Figure 20-12 and Figure 20-13 show the Protocol Threat Protection values for the XML Firewall and Web Service Proxy, respectively. (The MPGW does not have this area in its XML Threat Protection page, but you can specify the HTTP version on any FSH.) The choices for the WSP are the same ones we find for characterizing message types on the other services.

Earlier, we discussed how characterizing the message types could help to filter out some attacks, for example denying straight XML attacks because we have specified to accept only SOAP. These protocol settings are similar—specifying that only HTTP 1.1 connections allowed to the device will filter out any HTTP 1.0 traffic, which may be used on some of the antiquated systems that hackers around the world might target.

Protocol Threat Protection

Request HTTP Version

HTTP 1.1

Response HTTP Version

HTTP 1.1

Figure 20-12 XML Firewall Protocol Threat Protection.

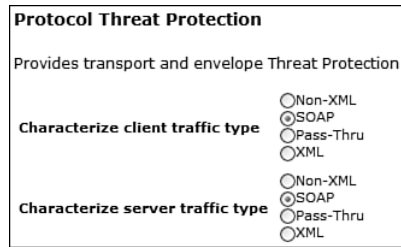


Figure 20-13 Web Service Proxy Protocol Threat Protection.

Message Tampering Protection

This section offers no configuration; it just suggests and acts as a reminder that you add a Validate action to your policies to validate messages against their schema to ensure they have no extra content or other schema violations.

We would like to take this opportunity to reinforce this advice. Schema validation is important, and in too many cases, we see it disabled either due to schema files being out of date or never created, or for fear of performance problems. Of course, schema validation is fundamentally useless unless you have strongly typed and well-defined schemas!

Because you have the massive processing power of DataPower at your disposal, feel free to do all those things you were afraid to do on your software-based systems. This includes schema validation—on both request *and* response messages! Many times when our security hats are on, we are too focused on the front door (incoming traffic), whereas our back door is wide open to attack. Often systems are harmed not by bad guys and gals, but by bad program code by partners and other intermediaries, and yes, sometimes even our own programmers. This is a far more frequent occurrence than being hacked. This is why we should use validation as often as possible. For example, after we transform messages, those transform results should be validated before sending them off to a backend or client that they could potentially crash.

Response validation also makes sure that only the data you want to leave your enterprise does so. Returning too many entries or fields that don't meet the schema could mean an unauthorized access, injection, or other type of data acquisition attack.

TIP — VALIDATE EVERYTHING!

Validate requests, responses, and any time a message is altered on the device. You never know when something might have gone wrong unless you do. Nobody wants to deal with the consequences of "unexpected results." And remember, GIGO (Garbage In, Garbage Out). Unless your schemas are well written and strongly typed, schema validation cannot help!

SQL Injection Protection

Earlier in this chapter, we gave some extensive examples (and a little humor) on SQL injection. Remember, this doesn't necessarily mean that your SOAP messages are passing around SQL statements (although we've seen this in the field). The attack is based on modifying normal input data that is headed to your backend systems, which will in many cases be used to formulate SQL commands. To prevent this, the inputs should be inspected for SQL commands that might have been appended to the input data in an attempt to get them to execute as well.

You can implement protection against this type of attack quite easily in DataPower. In fact, most or all the work is already done for you. Let's walk through an example implementation. To incorporate SQL injection protection, you start by adding a Filter to your Processing Policy and selecting the SQL-Injection-Filter.xsl stylesheet that resides in the device's store: directory, as shown in Figure 20-14.

Configure Filter Action

Basic **Advanced**

Input

Input: (auto) (auto) ▼

Options

Filter

store:/// SQL-Injection-Filter.xsl Upload... Fetch... Edit...

Processing Control File View... Var Builder *
Stylesheet Summary: Scan document for SQL Injection attacks.

Use WSDL Tool

Asynchronous ☐ on ☒ off

Output

Output: NULL NULL ▼

Delete Done Cancel

Figure 20-14 Filter action with SQL Injection Filter.

This stylesheet is worth opening and inspecting. There is a lengthy comment prologue that describes how the configuration works, and states a recommended two-step process for preventing SQL injection attacks:

1. Use the SQL injection filter included to capture all common data acquisition and destruction attempts.
2. Schema validate all response data, for example to ensure that if you are only expecting one row of data to return, that multiple rows are not returned. (As you have read, that is a symptom of a SQL injection attack!)

Upon inspection of the stylesheet, you will see that it loads a SQL patterns XML file. This stanza is shown in Listing 20-7.

Listing 20-7 SQL-Injection-Filter.xsl Patterns File Identification

```

<xsl:param name="dpconfig:SQLPatternFile" select="'store:///SQL-
Injection-Patterns.xml'"/>
<dp:param name="dpconfig:SQLPatternFile" type="dmFSFile" xmlns="">
  <display>SQL Injection Pattern File</display>
  <location>store:</location>
  <location>local:</location>
  <default>store:///SQL-Injection-Patterns.xml</default>
  <description>The file containing patterns to search for in order to
detect SQL injection attacks.</description>
</dp:param>

```

Of course, there is always the possibility that some of these patterns could be identical to legitimate SQL that you may be passing in your messages, which would not be good because those valid messages will be rejected! You are free to modify the included XML file and place a modified copy in the local: directory per the instructions in the XSL file. (These are much more detailed than the WebGUI Guide.) If you do this, you must modify the filter parameter `SQLPatternFile` that is show in Listing 20-7 by setting the parameter in either the firewall or Filter action. (The Add Parameter button appears on the Advanced tab of the Filter action.)

The filter searches through incoming messages and rejects any that match the regular expressions contained within. Let's look at the contents of the included XML file, as shown in Listing 20-8.

Listing 20-8 Top Portion of the SQL-Injection-Patterns.xml File

```

<pattern>
  <name>SQL LIKE% Match</name>
  <regex>'[\s]*[\%]+</regex>
</pattern>

<!-- SQL Escape Sequences -->
<pattern>
  <name>SQL Escape Sequence</name>

<regex>('[\s]*;|'[\s]*[\\])|'|'[\s]*[+]+'|'[\s]*[\*]+'|'[\s]*[=]+'</regex>
</pattern>

<!-- SQL keyword injection -->
<pattern type="element">
  <name>SQL Keyword Injection</name>
  <regex>^(insert|as|select|or|procedure|limit|order
by|asc|desc|delete|update|distinct|having|truncate|replace|handler|like)$</
regex>
</pattern>

```

```
<!-- MS SQL Commands -->
<pattern>
  <name>MS SQL Commands</name>
  <regex>\b(exec sp|exec xp)\b</regex>
</pattern>
```

The file starts out by looking for and rejecting some typical escape sequences used in SQL, such as the semicolon used to delimit commands or append new commands. Following that, certain keywords are rejected—such as update, delete, and truncate. The last stanza shown in our listing (the file is actually much longer with many additional examples) is one that is specific to Microsoft SQL Server.

You might notice that the pattern with the name SQL Keyword Injection has an attribute named “type” on the pattern element. This tells the stylesheet to search only elements. Correspondingly, if it is not present, or is set to type=global, both elements and attribute values will be scanned for offending values.

To gain a better understanding of how this works at runtime, let’s put together an example. We have configured a simple loopback XML Firewall with a Filter action set up as shown in Figure 20-14, using the out-of-the-box SQL Injection stylesheet and XML file. We pass in a file with a SQL command string concatenated to the data, as shown in Listing 20-9.

Listing 20-9 Input File with SQL Injection Attack

```
<?xml version='1.0' ?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Body>
    <book>
      <name>Moby Dick';SELECT * FROM BOOK_AUTHORS WHERE AUTHOR_NAME
LIKE '%KEROUAC%</name>
      <author>Herman Melville</author>
    </book>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

When the command is submitted, the client sees the rejection message, as shown in Figure 20-15. The resulting message does give evidence as to why the command has failed.

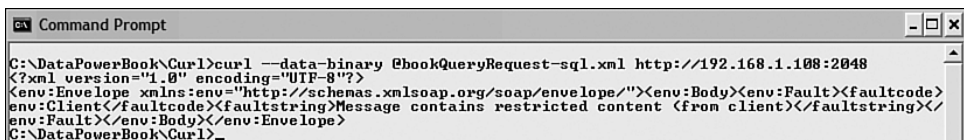


Figure 20-15 Client error message indicating restricted content.

For a more detailed analysis, the log messages for the firewall can be viewed. Figure 20-16 shows those log messages, with a detailed explanation of what has transpired and why the transaction was rejected. Notice it tells exactly what pattern has caused the rejection, in this case the one named SQL LIKE% Match, which is the first one shown in Listing 20-8.

```
xmlfirewall (XMLThreatTestFirewall): No match from processing policy 'XMLThreatTestFirewall' for code '0x00d30003'
xmlfirewall (XMLThreatTestFirewall): Rejected by filter; SOAP fault sent
xmlfirewall (XMLThreatTestFirewall): Generated error on URL 'http://192.168.1.108:2048/': <?xml version="1.0" encoding="UTF-8"?>
<env:Envelope
xmlns:env="http://schemas.xmlsoap.org/soap/envelope/"><env:Body><env:Fault><faultcode>env:Client</faultcode><faultstring>Message
contains restricted content (from client)</faultstring></env:Fault></env:Body></env:Envelope>
xmlfirewall (XMLThreatTestFirewall): Rejected by filter; SOAP fault sent
xmlfirewall (XMLThreatTestFirewall): request XMLThreatTestFirewall_request #1 filter: 'INPUT store:///SQL-Injection-Filter.xml' failed: Message
contains restricted content
xmlfirewall (XMLThreatTestFirewall): Execution of 'store:///SQL-Injection-Filter.xml' aborted: Message contains restricted content
xmlfirewall (XMLThreatTestFirewall): ***SQL INJECTION FILTER***: Message from 192.168.1.102 contains possible SQL Injection Attack of
type 'SQL LIKE% Match'. Offending content: '%', Full Message: '<?xml version="1.0" encoding="UTF-8"?> <SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"> <SOAP-ENV:Body> <book> <name>Moby Dick;SELECT * FROM
BOOK_AUTHORS WHERE AUTHOR_NAME LIKE "%KEROUAC%"</name> <author>Heman Melville</author> </book>
</SOAP-ENV:Body> </SOAP-ENV:Envelope>'.
xmlfirewall (XMLThreatTestFirewall): Reject set: Message contains restricted content
xmlfirewall (XMLThreatTestFirewall): Finished parsing store:///SQL-Injection-Patterns.xml
xmlfirewall (XMLThreatTestFirewall): Parsing document 'store:///SQL-Injection-Patterns.xml'
xmlfirewall (XMLThreatTestFirewall): Stylesheet URL to compile is 'store:///SQL-Injection-Filter.xml'
xmlfirewall (XMLThreatTestFirewall): Finished parsing http://192.168.1.108:2048/
xmlfirewall (XMLThreatTestFirewall): Parsing document 'http://192.168.1.108:2048/'
xmlfirewall (XMLThreatTestFirewall): rule (XMLThreatTestFirewall_request): selected via match 'XMLThreatTestFirewall' from processing policy
'XMLThreatTestFirewall'
xmlfirewall (XMLThreatTestFirewall): New transaction(conn use=1): POST http://192.168.1.108:2048/ from 192.168.1.102
```

Figure 20-16 DataPower log showing repulsion of a SQL injection attack.

XML Virus Protection

XML Virus Protection is used for checking messages and attachments for embedded viruses. The types of attacks that can be thwarted by this action are XML Virus Attacks, XML Encapsulation Attacks, Payload Hijack Attacks, and Binary Injection Attacks. Of course, DataPower does not have a built-in virus scanner, or you would have to constantly update it to include new virus signatures, much like you do for your workstation's virus scanner. (You have one of those, right?) The way this works in DataPower is that the device uses the Internet Content Adaptation Protocol (ICAP) to send the content to a virus scanning server, in most all cases the same one that is used to scan your email attachments. You have the option to reject transactions or strip out the attachment and let it proceed, if the virus scanner reports a problem.

In firmware versions prior to 3.6.1, this would be configured similarly to the SQL Injection Protection—by placing a Filter on your policy, assigning a stylesheet store:///Virus-ScanAttachments.xml, and assigning a value representing the URL of your virus scanning server to the SendTo parameter.

You may still see the instructions to configure virus protection this way on the XML Threat Protection tab, depending on what version of the firmware you are on. However, in 3.6.1 a new Anti-Virus action was introduced to make this process easier.

Figure 20-17 shows the Anti-Virus action configuration, which we arrived at by dragging an Advanced action to our request rule processing line and clicking the Next button.

Configure Anti-Virus Action

Basic **Advanced**

Input

Input INPUT INPUT

Options

Anti-Virus

Asynchronous ☐ on ☒ off

Anti-Virus Scan Type

☒ Scan Entire Message
☐ Scan All Attachments
☐ Scan Attachments by Content Type
☐ Scan Attachments by URI
☐ Scan by XPath Expression

* Save

ICAP Host Type

☐ Clam
☒ Symantec
☐ Trend Micro
☐ Webwasher
☐ Custom

* Save

Remote Host Name 192.168.1.110 * Save

Remote Port 1344 Save

Remote URI /AVSCAN?action=SCAN Save

Anti-Virus Policy

☐ Log
☒ Reject
☐ Strip

* Save

Anti-Virus Error Policy

☐ Log
☒ Reject
☐ Strip

* Save

Log Category xsiltmsg + ... Save

Output

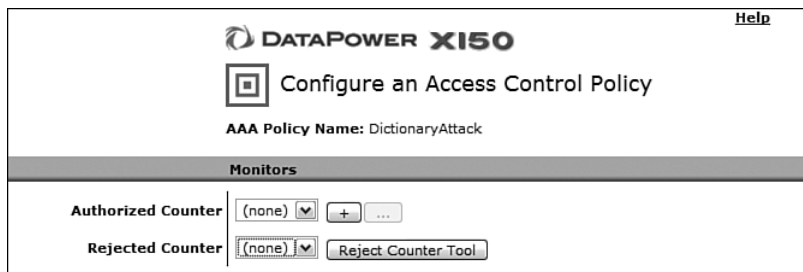
Output dptemp1 dptemp1

Delete Done Cancel

Figure 20-17 Configuration for the Anti-Virus action.

Dictionary Attack Protection

The Dictionary Attack Protection text on the XML Threat Protection pane explains that these are monitored through AAA actions, which makes sense since dictionary attacks are login-based. The policy can be set up to send notification messages to a log target, or put the offending client in “time out” (deny service) for a while. The text goes through some requirements for setting this up, but by far it’s easier to just do this from the Post Processing page of the AAA wizard, using the Reject Counter tool, as shown in Figure 20-18. Notice that this field does not have the traditional buttons next to it to add or edit the configuration! We will explain that in a moment.



DATAPOWER X150 [Help](#)

Configure an Access Control Policy

AAA Policy Name: DictionaryAttack

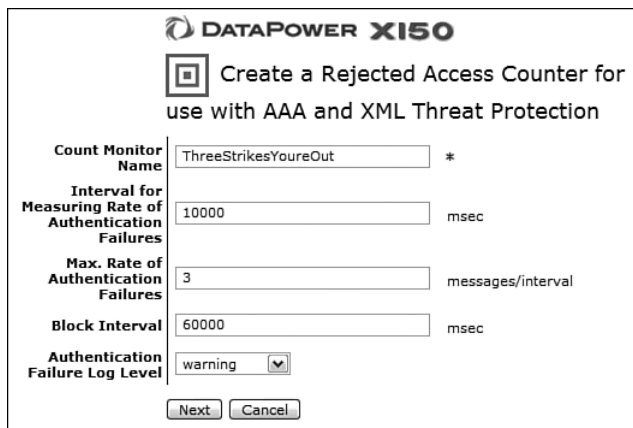
Monitors

Authorized Counter: (none) + ...

Rejected Counter: (none) Reject Counter Tool

Figure 20-18 The AAA Post Processing Reject Counter tool.

This counter functions regardless of the method chosen for identity extraction and authentication in the earlier pages of the AAA configuration. The configuration is fairly simple as shown in Figure 20-19. We have configured the policy such that if any client gives an invalid authentication more than three times within ten seconds, they may not try again for a minute, and a log message with the severity of warning is issued. Log targets could possibly be set up to send email to an administrative group, or fire an SNMP trap to the datacenter monitoring infrastructure. We covered many options for logging in Chapter 14, “Logging and Monitoring.”



DATAPOWER X150

Create a Rejected Access Counter for use with AAA and XML Threat Protection

Count Monitor Name: ThreeStrikesYoureOut *

Interval for Measuring Rate of Authentication Failures: 10000 msec

Max. Rate of Authentication Failures: 3 messages/interval

Block Interval: 60000 msec

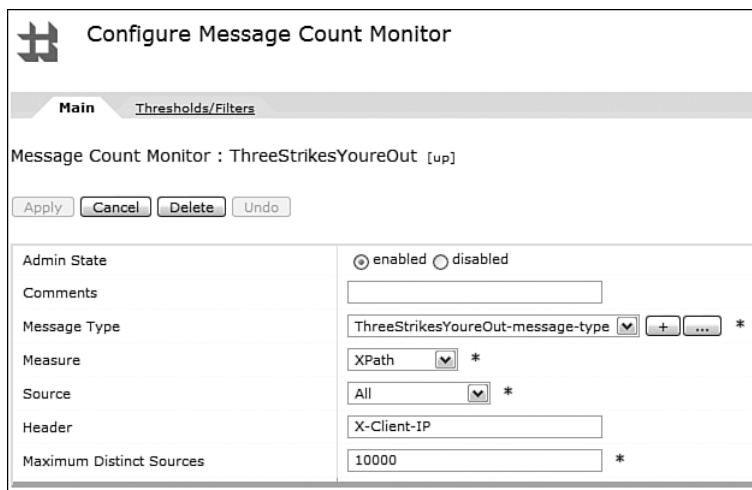
Authentication Failure Log Level: warning

Next Cancel

Figure 20-19 A Reject Counter configured for the three strikes rule.

Bear in mind that most dictionary attacks are automated, and the requests will fire in very quickly. Ours was configured somewhat loosely on the time values because we used it from a command shell with a batch file, rather than a sophisticated load generating script or tool. To prevent scripted dictionary attacks, you would set the threshold much lower for the time interval used for detection.

Let's have a look under the covers. What really happened when we built that Reject Counter tool, and why are there no buttons to edit it, in case we made a mistake? What really happens is that DataPower builds a Message Count Monitoring object for you. It uses the data that you have entered from the reject tool and adds defaults for other fields. To view or change this, you would have to go to Objects→Monitoring→Message Count Monitor in the Navigation menu of the WebGUI, and there you will find the object that you named (ThreeStrikesYoureOut in this case). Clicking on it reveals the configuration, as shown in Figure 20-20. Notice here that the X-Client-IP address is being used as a basis for counting—in other words more than three invalid login attempts from the same IP will trigger the time out and log message.



Configure Message Count Monitor	
Main Thresholds/Filters	
Message Count Monitor : ThreeStrikesYoureOut [up]	
Apply Cancel Delete Undo	
Admin State	<input checked="" type="radio"/> enabled <input type="radio"/> disabled
Comments	<input type="text"/>
Message Type	ThreeStrikesYoureOut-message-type <input type="button" value="+"/> <input type="button" value="..."/> *
Measure	XPath <input type="button" value="v"/> *
Source	All <input type="button" value="v"/> *
Header	X-Client-IP <input type="text"/>
Maximum Distinct Sources	10000 *

Figure 20-20 The Message Count Monitor object generated by the Reject Counter tool.

Moving over to the Thresholds/Filters tab, the content of which is shown in Figure 20-21, you may notice something suspicious. If you were to test this policy now, by sending four bad logins followed by a good one, the good one would succeed, and the log message and login denial period would not occur. This figure tells us exactly why. The default for the policy allows for “bursts” in the traffic. These might be acceptable in some message counters, but not for the type of Reject Counter on authentications that we are monitoring. You might want to change this default to zero. You should also configure the message count monitor that was created from the AAA policy to be active on your service, for example on the Monitors tab of an XML Firewall or Multi-Protocol Gateway.

Figure 20-21 The Thresholds/Filters settings for the ThreeStrikesYoureOut reject counter.

While we are on this pane, let's have a look at the Message Filter action object that has been built for us. Figure 20-22 shows what we expect—the transaction will be rejected, a log message issued at the level of warning, and a block interval of 60 seconds.

Figure 20-22 The Message Filter action object for the Reject Counter.

Using the Filter Action for Replay Attack Protection

Replay attacks can be prevented by using the Filter action. On the Advanced tab you will find the option to choose a Filter Method of “Replay Filter.” This exposes the option to choose a Replay Filter Type, which includes WS-Addressing Message ID, WS-Security Password Digest Nonce, and Custom XPath. This capability is discussed further in Chapter 19.

SLM Policies

Although service-level management (SLM) is normally thought of in performance, reliability or quality-of-service scenarios, it is also a good way to prevent DoS attacks. By the nature of what SLM does—rate-limiting and shaping traffic—it can be quite useful in thwarting or combating DoS attacks—at least to prevent them from reaching the backend servers. Figure 20-23 shows an SLM policy for a Web Service Proxy that kicks in for one particular backend Web service

operation when the traffic reaches 500 transactions per second. This particular policy is set up to use DataPower’s unique shaping algorithm, where transactions are queued on the device to introduce slight latencies to smooth out the curve of peak demand. This is covered in Chapter 10.

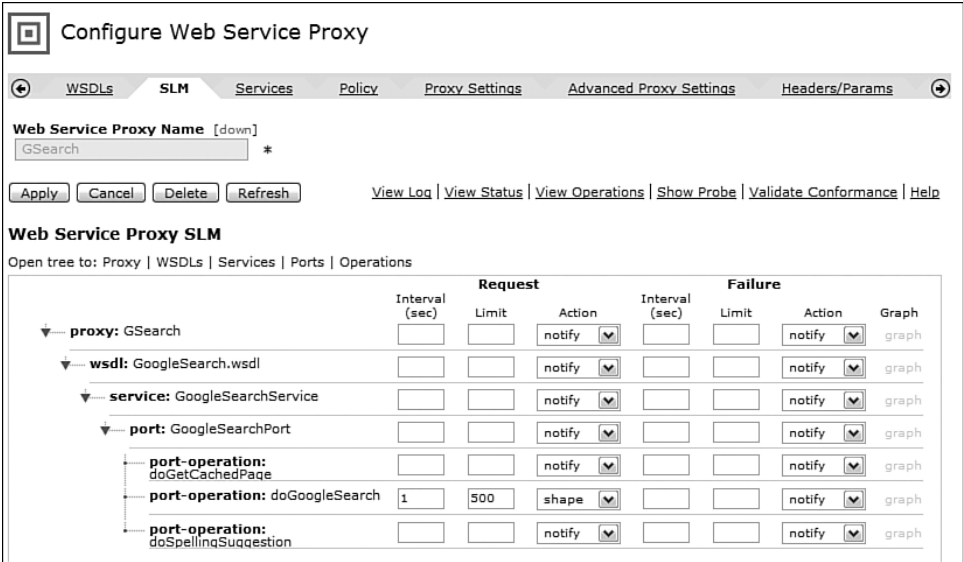


Figure 20-23 The SLM policy for a Web Service Proxy.

Summary

In this chapter, we discussed a wealth of different types of attacks that can be mounted using XML. We built on that to show the many ways that you can protect your systems from those attacks using the features of DataPower SOA appliances. Some of this is incredibly easy (in fact you get a great deal with no additional configuration whatsoever), and some of it takes some investigation into your message metrics, thought, and work on your part. In the end, the results always pay off. The cost of production outages and system compromises is generally staggering.

In conclusion, it’s important to note that even though the bad guys are out there, often we simply need to protect ourselves against malformed or rogue messages from our partners, clients, and even our own programmers. Many times the messages that crash our systems are simply a result of bad application code or transforms.

For this reason, we can’t stress strongly enough that the advice in this chapter be taken to heart—validate all request and response messages and the results of any transforms or message alterations. Keep the many settings in DataPower that are related to threat protection in mind and make it part of your application acceptance process to review and tune for these, just as you (hopefully) tune your applications and infrastructure for performance.