Chapter

Analyzing and Blocking Malicious Traffic Based on Geolocation

INFORMATION IN THIS CHAPTER:

- Research and Due Diligence
- Implementing a Solution
- Integrating with TMG

PRODUCTS, TOOLS, AND METHODS:

- MS SQL Server
- TMG
- Visual Studio (C#/Visual Basic for Applications)
- TMG Logging to SQL Server
- SQL CLR
- TMG Deny Rules

INTRODUCTION

Years back, while working on a research project involving SQL Slammer and the persistence of vulnerable installations around the globe, I started paying particular attention to the trending of attacks originating from and associated with vulnerabilities within different countries. In this particular case, China headed the list with the most attacks originating within its borders. This piqued my interest not only in quantifying attacks targeted at age-old vulnerabilities, but also in looking further into instances where an individual country was more exposed than others, and where more attacks were sourced for any particular vulnerability.
Further research increased my interest and prompted the writing of an article\(^1\) where I floated the concept of designing security controls around the actual source of the attacks rather than the vector that an attack was trying to leverage. I have published different aspects of this overall approach around the Internet in the past few years, and have continued to focus on how applicable this control would be in a production network, mostly by way of collecting as much data as I could and analyzing it in different ways as the opportunities presented themselves. As such, I have included some of that new data here along with some interesting code and tools to freshen up the project and present it here.

**NOTE**

This is where my obligatory disclaimer comes in. The aforementioned article prompted its share of contemptuous and critical e-mails where people thought I had some position about China specifically. I do not. I do not care if the traffic is coming from Disneyland. If it is attacking my network, I am going to block it. As such, nothing in the following chapter is motivated by any political, religious, racial, or other ideal or prejudice. Processes discussed and opinions reached are based on technical and statistical research. No other bias is intended nor should be inferred.

**RESEARCH AND DUE DILIGENCE**

When you are considering the implementation of a security control, having a basis of research from which to make decisions is extremely important, if not critical. Of course, what a basis consists of is up to you. Personally, my mind was made up from the initial results of my Slammer traffic captures. I saw that China sourced a level of malicious traffic that I did not have any production reasons to process, so I blocked the entire country. However, it would have been irresponsible of me to move forward without performing due diligence in research and data collection. Based on the analysis of many different attacks, vulnerabilities, and overall junk traffic, this decision has turned out to be valuable to me. Even with substantial data to support the quantification of traffic from China and other countries of interest, I felt like I had to better qualify the statistics by getting data from sources where actual attack information was being compiled. In other words, just knowing how much traffic was coming from countries I was concerned with was not enough; I wanted to know what the content of the traffic itself was, and what current trend of threat those attacks represented. After some conversations with other colleagues in the industry, I was fortunate enough to get in touch

\(^1\)www.symantec.com/connect/articles/blocking-traffic-country-production-networks
with some nice folks with the Honeynet Project, a leading international security research organization, who provided me with data for April of 2009 of attacks against a honeypot in Hawaii. There were almost 13,000 attacks categorized by type, such as mass mailing worms, attacks against P2P networks, protocol anomalies, and Denial of Service (DoS) attacks. They were also rated by severity and impact. Low-severity and impact attacks were numerous and interesting, but I wanted to focus on the high-severity attacks which were likely to succeed based on Honeynet’s rating. I further limited the attacks to DoS attacks to level the playing field against OS types, patch levels, and so on. I ended up with 2,527 high-risk/success attacks that I subsequently broke down by country of source, as shown in Figure 3.1.

I found the results compelling. They are ranked highest to lowest and read left to right, so the USA is first, China is second, Japan is third, and so on. Almost 30 percent of the attacks were sourced from China, with over 50 percent coming from countries that I might very well be able to block. I have to say that I did feel a bit sorry for France, who could only muster a single successful attack, but I will leave any further significance of that statistic for you to infer.

“Likely Successful” Attacks

- United States
- China
- Japan
- Czech Republic
- Korea, Republic of
- Iraq
- Canada
- Antigua and Barbuda
- Philippines
- United Kingdom
- Romania
- France

2527 High-Risk Attacks, Hawaii Honeynet, April 2009

FIGURE 3.1 Likely Successful High-Risk Attacks by Country (Ratings by Honeynet)
Research Findings

This is the type of data that allows you to make intelligent decisions about controls that can be put around traffic. A control illustrates direct value when 30 percent of malicious traffic, traffic actually qualified as a would-be successful attack, can be significantly reduced just by geographically filtering traffic at your border.

In June of 2008, I published a paper on www.SecurityFocus.com providing some initial figures for various standard traffic patterns like HTTP and Simple Mail Transfer Protocol (SMTP) from different countries as well. That research also provided some interesting statistics around sources of SMTP, HTTP, FTP, SSH, and so on. During the preparation of this book, I wanted to be able to work with current data for examples of parsing and manipulating log data, so I initiated a full traffic capture and analysis project on July 19, 2010, which ran uninterrupted until February of 2011. Hammer of God is just a simple little domain I own, but I have had it for over a decade, so I get my fair share of spam just like everyone else. And though I have got various implementations of spam filtering, I wanted to see just how much SMTP traffic I was getting. Figure 3.2 shows the results. I was surprised by this research as well.

There is no business or personal reason for me to get e-mail from the Russian Federation. Yet, in about seven months, I blocked over 1 billion SMTP connection attempts to my network. And of the countries listed, I really only need e-mail from a few of them. In total, I blocked over 9 billion SMTP connections alone to my relatively tiny network. Those packets never even made it through to my mail gateway; they were immediately and silently dropped at the border.

In my opinion, this level of research and analysis is more than sufficient to warrant consideration of implementation into production environments where appropriate.

IMPLEMENTING A SOLUTION

Some of you do not have a business model that supports country blocking and some of you do. Irrespective of your disposition, I wanted to outline a process by which you could leverage a few technologies from which to perform data analysis, and to have the ability to take action should you decide to implement this control. It may be in the form of blocking SMTP from a particular set of countries, or web traffic from others, or blacklisting all traffic from any particular source.

So we will now get started.
FIGURE 3.2 Hammer of God’s Top 20 SMTP Sources by Country (Spelling from Original Source)

<table>
<thead>
<tr>
<th>SourceCountry</th>
<th>DstPort</th>
<th>Count</th>
</tr>
</thead>
<tbody>
<tr>
<td>Russian Federation</td>
<td>25</td>
<td>1080700</td>
</tr>
<tr>
<td>India</td>
<td>25</td>
<td>909634</td>
</tr>
<tr>
<td>Brazil</td>
<td>25</td>
<td>572700</td>
</tr>
<tr>
<td>Viet Nam</td>
<td>25</td>
<td>539937</td>
</tr>
<tr>
<td>Ukraine</td>
<td>25</td>
<td>466390</td>
</tr>
<tr>
<td>Indonesia</td>
<td>25</td>
<td>374473</td>
</tr>
<tr>
<td>Korea Republic of</td>
<td>25</td>
<td>336197</td>
</tr>
<tr>
<td>China</td>
<td>25</td>
<td>319347</td>
</tr>
<tr>
<td>France</td>
<td>25</td>
<td>232864</td>
</tr>
<tr>
<td>Romania</td>
<td>25</td>
<td>198408</td>
</tr>
<tr>
<td>Colombia</td>
<td>25</td>
<td>194663</td>
</tr>
<tr>
<td>Argentina</td>
<td>25</td>
<td>189289</td>
</tr>
<tr>
<td>Pakistan</td>
<td>25</td>
<td>181994</td>
</tr>
<tr>
<td>United States</td>
<td>25</td>
<td>177374</td>
</tr>
<tr>
<td>Taiwan, Republic of China (ROC)</td>
<td>25</td>
<td>170022</td>
</tr>
<tr>
<td>Italy</td>
<td>25</td>
<td>166182</td>
</tr>
<tr>
<td>Spain</td>
<td>25</td>
<td>152217</td>
</tr>
<tr>
<td>Kazakhstan</td>
<td>25</td>
<td>145305</td>
</tr>
</tbody>
</table>
In simple terms, we want to capture traffic, figure out what country it came from, analyze it to meet our needs, and then form rules to take the appropriate actions we want. In this example, I am using TMG, which already has logging capabilities in place for all aspects of IP traffic we need. The most obvious project dependency we have is a source for country-by-country IP ranges in order to identify the country based on IP. In the beginning, I spent a substantial amount of time aggregating IP source data from several sources until I decided to use a single source of data from WebNet77, but your sources are up to you.

Now we shall get into the tech of how to make this happen. Figure 3.3 illustrates an overall workflow diagram of our process to work from.

**Log Traffic**

Before we tackle figuring out how to identify the traffic source, we need to ensure that we are logging the raw data in a suitable environment. As I try to do whenever possible, this is a case where we can build upon the logging setup and design already discussed in the book. Please see the first chapter, “Securely Writing Web Proxy Log Data to Structured Query Language (SQL) Server and Programatically Monitoring Web Traffic Data in Order to Automatically Inject Allow/Deny Rules into Threat Management Gateway (TMG),” for complete instructions on how to get TMG logging in a manner conducive to this project if you have not already.

Figure 3.4 shows WebNet77’s comma delimited download.

**NOTE**

An IPv4 address is simply a dot notation of octets that represent an integer value from 0 to 4,294,967,295 or 0.0.0.0 to 255.255.255.255, or x00000000 to x0FFFFFF. Hex dotted notation works as well in the form of FF.FF.FF.FF, which directly converts to 255.255.255.255, which in turn converts to 4,294,967,295 decimal in the form of 256^3.256^2.256^1.256^0. This is important to know when you begin to work with the logging of IP addresses and the subsequent mapping of an IP address to a record containing a range of IPs as in Figure 3.4.

**Data Functions**

In human-readable reports, the IP address is typically in dotted notation, while system-based log values and database fields typically work with integers. For instance, in both ISA Server and TMG, IP address records in a monitoring

---

2http://software77.net/geo-ip/
3www.subnetmask.info/
session will be displayed in dot notation even though the address is being stored differently: ISA Server logs store the IP as an integer value while TMG stores it as a *uniqueidentifier* data type. See Figure 3.13 for further information.

You will need dotted notation as you look at reports and later when you programmatically create TMG computer set objects, but you will use integers
when you log and compare records. I will of course include some code I have written to do this for you, but first we will have a quick math lesson so we know what is happening as we convert back and forth. Borrowing from examples used in one of my previous articles, we will take the IP address 203.83.16.1, which was last seen somewhere in Papua New Guinea. Starting with the base octet form of $256^3 \cdot 256^2 \cdot 256^1 \cdot 256^0$, we will extrapolate each decimal equivalent out respectively, resulting in $16777216 \cdot 65536 \cdot 256 \cdot 0$, as odd as that may look. With that base, we multiply each octet by the base and add them together, yielding:

$$(16,777,216 \cdot 203) + (65,536 \cdot 83) + (256 \cdot 16) + (1)$$

which equals

$$(3405774848) + (5439488) + (4096) + (1)$$

This finally results in the integer 3,411,218,433 (shown with comma separators obviously). To reverse the process and create a dotted octet notation from an integer, we cast the integer as a binary SQL data type (hex), carve it up into octets, and convert back to decimal separated by a period. 3411218433 converted to hex is xCB531001, dotted out to CB.53.10.01, and finally to 203.83.16.1, which is a good bit easier to do than the other way around.

Before we can use this, we need to import the IpToCounty data into SQL. This is done easily enough by opening the CVS file downloaded from WebNet77 in Excel, saving it as a workbook, and then importing it into the log database similar to that described in figure 3.5. The IpToCountry table already has the big integer (BIGINT) data types specified for the

![Figure 3.4 WebNet77 Sample Comma Delimited IpToCountry Rows](image-url)
integer values, so we just have to add a couple of fields for the dot notation as shown in Figure 3.5.

**NOTE**

Be sure that when you create this table you configure `BIGINT` data types, and not just `INT` data types. The INT data type can only store values 2,147,483,648 through 2,147,483,647 and requires four bytes. These four bytes could represent 4294967294 integers, but only if they are unsigned. The INT will only let you store up to IP 127.255.255.255, which will bite you moving forward. BIGINT might take up twice as much storage, but it is easier to work with. Of course, if you are worried about storage, then you will have to work with four-byte binary data or muck about with negative numbers.

With the IpToCountry data imported into our SQL log database and the BegIP and EndIP character fields added, we get a list of all the Papua New Guinea records, similar to what is shown in Figure 3.6.

Note that the BegIP and EndIP columns are null, as they should be. We now need to update those fields by converting the BegIPLong and EndIPLong integer fields to dot notation. At this point, we will convert the manual process we did earlier into a SQL scalar-value function to do it for us. This function takes a BIGINT input parameter, converts it to variable length binary data (varbinary), parses it out as described earlier, and returns the dot notation.
Figure 3.7 shows the successful results of the function.

With this simple function, we can now update all 100,000+ records of our IpToCountry table with this one SQL command, as shown in Figure 3.8.

At this point, you may be asking yourself why we need the dot notation IP address in this table. We have got the beginning and ending integer equivalence, and if our base logging data type is based on integers in ISA Server (again, TMG uses a uniqueidentifier, which we will discuss), why would we need the dot notation? We need the dot notation because at some point we will be exporting this data to TMG in the form of a computer set, and the computer set input mechanism is by beginning and ending IP address in dot notation, not integer. So this step
is required if we are to build sets from this data. It is also human-readable, which helps as you scan through records trying to manually match dot notation entries to a range. And to me, it just looks more natural to have the dot notation included, as demonstrated in Figure 3.9.

Again, the dot notation does not help us match a country to a record; the BegIPLong and EndIPLong integer values do that. But as far as having a table that allows us to look up a country based on its (converted) IP address, we have everything we need.

**Linking the Data**

For those of you not explicitly familiar with SQL Server and T-SQL in general, the query method used to find records within this table structure may be a bit counterintuitive, but it is fairly straightforward. We have the beginning integer and ending integer for a given range along with the country name in the table. Given an integer representation of an IP (in this example, 3411218433), we can determine which country it is from with a query or procedure that finds the records where the value is between the two ranges, as Figure 3.10 illustrates.

We simply provide the value, and tell SQL to find the record where that value is between the BegIPLong and EndIPLong values. It is not exactly
the most performance-optimizing structure we could use, but given our
options, it is what we have to work with.

So we can take an integer and convert it to an IP, but now we need to be able to
parse out a dot notation IP address and convert it to an integer. This is for cases
where we only have dotted IP or when we just want to test. In order to do this
efficiently, we will build another function and then use that function inside
Implementing a Solution

**FIGURE 3.9** List of Papua New Guinea IP Ranges Including Dot Notation

```sql
SELECT BegIP, EndIP, BegIPLong, EndIPLong, CountryName
FROM IpToCountry
WHERE CountryName LIKE 'Papua%'
```

<table>
<thead>
<tr>
<th>BegIP</th>
<th>EndIP</th>
<th>BegIPLong</th>
<th>EndIPLong</th>
<th>CountryName</th>
</tr>
</thead>
<tbody>
<tr>
<td>14.192.72.0</td>
<td>14.192.75.255</td>
<td>247482368</td>
<td>247483391</td>
<td>Papua New Guinea</td>
</tr>
<tr>
<td>27.122.16.0</td>
<td>27.122.31.255</td>
<td>460984320</td>
<td>460988415</td>
<td>Papua New Guinea</td>
</tr>
<tr>
<td>119.252.224.0</td>
<td>119.252.239.255</td>
<td>2013061120</td>
<td>2013065215</td>
<td>Papua New Guinea</td>
</tr>
<tr>
<td>124.240.192.0</td>
<td>124.240.223.255</td>
<td>2096152576</td>
<td>2096160767</td>
<td>Papua New Guinea</td>
</tr>
<tr>
<td>180.150.252.0</td>
<td>180.150.255.255</td>
<td>3029793792</td>
<td>3029794815</td>
<td>Papua New Guinea</td>
</tr>
<tr>
<td>202.0.80.0</td>
<td>202.0.80.255</td>
<td>3389018112</td>
<td>3389018367</td>
<td>Papua New Guinea</td>
</tr>
<tr>
<td>202.1.32.0</td>
<td>202.1.63.255</td>
<td>3389071360</td>
<td>3389079551</td>
<td>Papua New Guinea</td>
</tr>
<tr>
<td>202.1.240.0</td>
<td>202.1.255.255</td>
<td>3389124608</td>
<td>3389128703</td>
<td>Papua New Guinea</td>
</tr>
<tr>
<td>202.52.133.0</td>
<td>202.52.133.255</td>
<td>3392439552</td>
<td>3392439807</td>
<td>Papua New Guinea</td>
</tr>
<tr>
<td>202.58.128.0</td>
<td>202.58.131.255</td>
<td>3392831488</td>
<td>3392832511</td>
<td>Papua New Guinea</td>
</tr>
<tr>
<td>202.61.0.0</td>
<td>202.61.0.255</td>
<td>3392995328</td>
<td>3392995583</td>
<td>Papua New Guinea</td>
</tr>
<tr>
<td>202.95.192.0</td>
<td>202.95.207.255</td>
<td>3395272704</td>
<td>3395276799</td>
<td>Papua New Guinea</td>
</tr>
<tr>
<td>202.165.192.0</td>
<td>202.165.207.255</td>
<td>3399860224</td>
<td>3399864319</td>
<td>Papua New Guinea</td>
</tr>
<tr>
<td>202.171.240.0</td>
<td>202.171.247.255</td>
<td>3400265777</td>
<td>3400267775</td>
<td>Papua New Guinea</td>
</tr>
<tr>
<td>203.83.16.0</td>
<td>203.83.23.255</td>
<td>3411218432</td>
<td>3411220479</td>
<td>Papua New Guinea</td>
</tr>
</tbody>
</table>

**FIGURE 3.10** Find Country Based on Integer IP Address Integer Value

```sql
SELECT CountryName
FROM IpToCountry
WHERE 3411218433 BETWEEN BegIPLong AND EndIPLong
```

<table>
<thead>
<tr>
<th>CountryName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Papua New Guinea</td>
</tr>
</tbody>
</table>

---

*Note: The SQL queries and results are for illustrative purposes and may not reflect actual data.*
CREATE FUNCTION [dbo].[fn_SQLString2Long]
(@ip CHAR(15))
RETURNS BIGINT
AS
BEGIN
DECLARE @rv BIGINT,
@o1 BIGINT,
@o2 BIGINT,
@o3 BIGINT,
@o4 BIGINT,
@base BIGINT
SELECT
@o1 = CONVERT(BIGINT, PARSENAME(@ip, 4)),
@o2 = CONVERT(BIGINT, PARSENAME(@ip, 3)),
@o3 = CONVERT(BIGINT, PARSENAME(@ip, 2)),
@o4 = CONVERT(BIGINT, PARSENAME(@ip, 1))
IF (@o1 BETWEEN 0 AND 255)
AND (@o2 BETWEEN 0 AND 255)
AND (@o3 BETWEEN 0 AND 255)
AND (@o4 BETWEEN 0 AND 255)
BEGIN
SET @rv = (@o1 * 16777216) +
(@o2 * 65536) +
(@o3 * 256) +
(@o4)
END
ELSE
SET @rv = -1
RETURN @rv
END
a stored procedure. This is where we will take apart the dot notation’s 256\(^4\).256\(^3\).256\(^2\).256\(^0\) decimal equivalence to derive the integer shown in Code Sample 3.2.

It seems like quite a bit of code just to convert an IP to an integer, and it is, but that happens to be the way T-SQL works. The process required to convert TMG’s uniqueidentifier to an integer is even more complicated, but we have better solutions for that coming up.

Testing the function, we should see results similar to those shown in Figure 3.11.

In the same way that we retrieved the Papua New Guinea record by explicitly providing the integer value in Figure 3.10, we can now substitute the function to find the same record, as demonstrated in Figure 3.12.

**Process Source Country**

Now that the two-way conversion functionality exists, we can easily get whatever country-based information we want from the logs based on what our needs are. However, now is the time for you or your database administrators to make some design decisions. Deriving the country from stored integer values in reports or data sets is going to have significant performance issues. If you only have the raw integer IP data stored, and if every time you wish to reference the country you have to derive it by including the country
lookup in a subquery, then each record in your result set will have to be evaluated and matched. Depending on your environment, that may be acceptable, but for my purposes, no.

I opted for the data separation and parsing model where I created a customized table containing the relevant log data I wished to retain, and populated that table with data and associated country lookup information via a scheduled SQL job that runs each night. Since my logs collect all inbound and outbound connections, there is a tremendous amount of traffic that does not relate to this particular function. However, I want to keep that data. So, I designed a process where the data I want was taken out of the TMG firewall logs, posted to my reporting table, parsed out, and updated. That way I can do whatever I want with that table without affecting other data sets. One thing to remember is that if you change the actual TMG WebProxyLog and FirewallLog table structures stored in SQL, you will have to change the procedures TMG uses to post data to those tables during the logging process. I recommend against that. Try to keep the logging process as simple and uncomplicated as you can. Actually changing how the data gets logged can be risky and you could introduce problems that you won’t be aware of, which could include failing to log at all. This process could end up being a good example of how trying to implement a valuable security feature ended up causing more problems than you had in the first place—and nobody wants to be That Guy.

INTEGRATING WITH TMG

Now is a good time to address a TMG-specific logging element we must solve for. Again, while we have everything we need to actually create TMG-based Computer Set objects to block or allow traffic at the firewall, the current functions we have implemented at this point can only support linkage to ISA Server log formats since ISA Server logs the integer of the IP address and TMG logs IP as a uniqueidentifier data type, or a Global Unique Identifier (GUID) as it is also generally referred to. The purpose of the GUID function is to be able to create a data element that is guaranteed to be a universally unique value no matter who generates it or when, which is done by using CPU clock and network interface controller variables. In that sense, it is really a UUID, but it is called global instead of universal. The part we care about here is a 16-byte (stored) binary value formatted field in the format of 00000001-FFFF-0000-0000-000000000000. TMG presumably uses this new format to support IPv6 logging, but that is not for me to say. Regardless, it is a new logging format and we have to be able to convert it to both integer and dotted IP in order to leverage our required functionality. Even though a uniqueidentifier data type is used, TMG is only storing
the IP, which is neither a unique ID nor a GUID. However, for simplicity’s sake, I will refer to the format as GUID from here on out to avoid confusion.

Decisions, Decisions . . .

As is typically the case, we will see that the decisions we are about to make regarding something as inane as converting data types can have an impact on our overall security. I think you should pay particular attention to these next couple of steps since one contains a simple nuance—and a perfectly valid command structure—which could weaken our security posture.

First, you should treat this like you would any other development project and get down to the business of converting the data. An actual GUID would use the entire data type space with a value like 6F9619FF-8B86-D011-B42D-00C04FC964FF. However, when TMG logs the IPv4 address with a GUID data type, it only uses the first 4 bytes of field for the IP, sets the second 2 bytes to FFFF, and fills the remaining 10 bytes with 0s. Our earlier example of 203.83.16.1, which is CB531001 in hex, would be logged as CB531001-FFFF-0000-000000000000 in the TMG GUID format.

Figures 3.13a and 3.13b show a comparison of ISA Server and TMG logs for IP address storage.

As you can see, insofar as TMG and the IP address are concerned, we only have to concern ourselves with the first four bytes of the GUID. To be pedantic, we are actually going to be working with the first eight characters of the GUID as string nvarchar data when we convert it and not actually the first four bytes of data, because the GUID field itself is a uniqueidentifier data type and its binary output is different. If you look at the binary output of the GUID field value of 6F9619FF-8B86-D011-B42D-00C04FC964FF, as stored in SQL, you will see 0xFF19966F868B11D0B42D00C04FC964FF. You will notice the reverse (little endian\(^4\)) storage of the first eight bytes which is something you might want to be aware of in the future. But for our purposes this does not matter since the output of the GUID is text in this example.

\(^4\)Little endian stores low order bits in the lowest memory address, or little end, first. Big endian is the opposite.
From here, we just need to parse out the first four bytes of the GUID, and convert that to an integer in order to find the source country. If you want to view the dot notation, then you will have to write a converter for that as well. Of course, we could just convert the GUID to an integer and use our existing fn_SQLLong2String function to convert it again.

As most people do, we would probably do a search on the Internet for a solution that someone else has already done the legwork on. Before long we would come across the public domain example shown in Code Sample 3.3.

```sql
CREATE FUNCTION [dbo].[fnIpAddressToText] (
    @Ipv6Address [uniqueidentifier]
) RETURNS varchar(40) AS
BEGIN
    DECLARE @strInAddress varchar(40)
    DECLARE @strOutAddress varchar(40)
    SET @strInAddress = LOWER(CONVERT(varchar(40), @Ipv6Address))
    SET @strOutAddress = ''
    IF (SUBSTRING(@strInAddress, 10, 4) = 'ffff')
    BEGIN
        -- ipv4 (hex to int conversion)
        DECLARE @IsNum int, @ZERO int, @IsAlpa int
        set @ZERO = ASCII('0')
        set @IsNum = ASCII('9')
        set @IsAlpa = ASCII('a') - 10
        DECLARE @intH int, @intL int
        SET @intH = ASCII(SUBSTRING(@strInAddress, 1, 1))
        IF (@intH <= @IsNum) SET @intH = @intH - @ZERO ELSE SET @intH = @intH - @IsAlpa
        SET @intL = ASCII(SUBSTRING(@strInAddress, 2, 1))
    END
    RETURN @strOutAddress
END
```

**CODE SAMPLE 3.3** `fnIpAdressToText`: Public Example of Converting TMG GUID to Dotted IP

---

*Multiple sources found.*
IF (@intL <= @IsNum) SET @intL = @intL - @Zero ELSE SET @intL = @intL - @IsAlpa
SET @strOutAddress = CONVERT(varchar(3), @intH * 16 + @intL) + '.'

SET @intH = ASCII(SUBSTRING(@strInAddress, 3, 1))
IF (@intH <= @IsNum) SET @intH = @intH - @Zero ELSE SET @intH = @intH - @IsAlpa
SET @intL = ASCII(SUBSTRING(@strInAddress, 4, 1))
IF (@intL <= @IsNum) SET @intL = @intL - @Zero ELSE SET @intL = @intL - @IsAlpa
SET @strOutAddress = @strOutAddress + CONVERT(varchar(3), @intH * 16 + @intL) + '.'

SET @intH = ASCII(SUBSTRING(@strInAddress, 5, 1))
IF (@intH <= @IsNum) SET @intH = @intH - @Zero ELSE SET @intH = @intH - @IsAlpa
SET @intL = ASCII(SUBSTRING(@strInAddress, 6, 1))
IF (@intL <= @IsNum) SET @intL = @intL - @Zero ELSE SET @intL = @intL - @IsAlpa
SET @strOutAddress = @strOutAddress + CONVERT(varchar(3), @intH * 16 + @intL) + '.'

SET @intH = ASCII(SUBSTRING(@strInAddress, 7, 1))
IF (@intH <= @IsNum) SET @intH = @intH - @Zero ELSE SET @intH = @intH - @IsAlpa
SET @intL = ASCII(SUBSTRING(@strInAddress, 8, 1))
IF (@intL <= @IsNum) SET @intL = @intL - @Zero ELSE SET @intL = @intL - @IsAlpa
SET @strOutAddress = @strOutAddress + CONVERT(varchar(3), @intH * 16 + @intL)
END
ELSE
BEGIN
-- ipv6
SET @strOutAddress = @strOutAddress + SUBSTRING(@strInAddress, 1, 4) + ':'
+ SUBSTRING(@strInAddress, 5, 4) + ':'
+ SUBSTRING(@strInAddress, 10, 4) + ':'
+ SUBSTRING(@strInAddress, 15, 4) + ':'
+ SUBSTRING(@strInAddress, 20, 4) + ':'
+ SUBSTRING(@strInAddress, 25, 4) + ':'
+ SUBSTRING(@strInAddress, 30, 4)
END
-- guid sample '6F9619FF-8B86-D011-B42D-FFF34FC964FF'
RETURN @strOutAddress
END

Code Sample 3.3 looks good, and in testing we see that it works. You might also notice that the developer used the exact same example GUID that I have used, which means that they did their homework first and snagged it from the MS Developer Network (MSDN) example you find when searching for the GUID uniqueidentifier. It also goes the extra step and parses out the full GUID text into IPv6 colon notation which is an added bonus.
While the functions in Figure 3.14 may work just fine, a developer may find it too complicated and processor intensive. There is also the overhead of the developer figuring out exactly what someone else did and why they did it so they can not only fully understand the function, but also be able to test and troubleshoot it. I know that does not always happen, but it should.

**Keeping It Simple(r)**

The "Keep It Simple, Stupid" model would rather rudely dictate that we choose a simpler and easier-to-understand solution. Since we only need the IPv4 address, and all we care about is the first eight characters, the code shown in Code Sample 3.4 would do what we need more efficiently.

```sql
CREATE PROCEDURE [dbo].[sp_UID2IP]
    @UIDLong uniqueidentifier
AS
BEGIN
    SET NOCOUNT ON;
    declare @UID nvarchar(10),
            @Hex1 nvarchar(5),
            @Hex2 nvarchar(5),
            @Hex3 nvarchar(5),
            @Hex4 nvarchar(5),
            @SQL nvarchar(1000),
            @Param nvarchar(100),
            @IP nvarchar(16)

    set @UID = LEFT(@UIDLong,8)
    select
        @Hex1 = '0x' + substring(@UID,1,2), @Hex2 = '0x' + substring
                         -- (@UID,3,2),
        @Hex3 = '0x' + substring(@UID,5,2), @Hex4 = '0x' + substring
                         -- (@UID,7,2)

    select dbo.fnIpAddressToText('6F9619FF-8B86-D011-B42D-00C04FC964FF'),
          dbo.fnIpAddressToText('ADAOC39B-FFFF-0000-0000-000000000000')
```

**FIGURE 3.14** Example IPv6 and IPv4 Output of the fnIpAddressToText Function

While the functions in Figure 3.14 may work just fine, a developer may find it too complicated and processor intensive. There is also the overhead of the developer figuring out exactly what someone else did and why they did it so they can not only fully understand the function, but also be able to test and troubleshoot it. I know that does not always happen, but it should.

**Keeping It Simple(r)**

The “Keep It Simple, Stupid” model would rather rudely dictate that we choose a simpler and easier-to-understand solution. Since we only need the IPv4 address, and all we care about is the first eight characters, the code shown in Code Sample 3.4 would do what we need more efficiently.

```sql
CREATE PROCEDURE [dbo].[sp_UID2IP]
    @UIDLong uniqueidentifier
AS
BEGIN
    SET NOCOUNT ON;
    declare @UID nvarchar(10),
            @Hex1 nvarchar(5),
            @Hex2 nvarchar(5),
            @Hex3 nvarchar(5),
            @Hex4 nvarchar(5),
            @SQL nvarchar(1000),
            @Param nvarchar(100),
            @IP nvarchar(16)

    set @UID = LEFT(@UIDLong,8)
    select
        @Hex1 = '0x' + substring(@UID,1,2), @Hex2 = '0x' + substring
                         -- (@UID,3,2),
        @Hex3 = '0x' + substring(@UID,5,2), @Hex4 = '0x' + substring
                         -- (@UID,7,2)

    select dbo.fnIpAddressToText('6F9619FF-8B86-D011-B42D-00C04FC964FF'),
          dbo.fnIpAddressToText('ADAOC39B-FFFF-0000-0000-000000000000')
```

**CODE SAMPLE 3.4** sp_UID2IP: T-SQL Procedure to Convert TMG GUID to Dotted IP
If we test the output, we get the results shown in Figure 3.15.

In this example, we take advantage of embedded CONVERT directives to convert the embedded converted values directly into a dotted IP address. Since we know that we are working with hex data that we wish to convert to integers to represent each octet, we can leverage the syntax of prepending 0x to character data to directly convert hex to integer. In other words, we may think of CB as hex data that represents 203, however to SQL it is just a couple of characters. Telling SQL to convert 0xCB to an integer explicitly tells it to treat the character data as hexadecimal input. No matter how geeky it may sound, little things like that make me happy. In fact, I like it so much (I am role playing now) that I do not mind concatenating string data together into the @SQL variable so that I can build a customized string to execute with the Sp_executesql system stored procedure. Not only is it flexible and fast, but prepending 0x to character data in a one-line, doubled-down back-to-back series of convert statements wins me geek points.

But it also opens up the possibility that a SQL injection vulnerability could be introduced since we are ultimately building a string variable from concatenating character data together to form an executable SQL statement. This example actually mitigates SQL injection due to the explicit declaration of the input parameter as a uniqueidentifier data type. This means

```sql
Set @SQL = N'Select
CONVERT(nvarchar(3),(CONVERT(int,' + @Hex1 + '))) + CHAR(46) +
CONVERT(nvarchar(3),(CONVERT(int,' + @Hex2 + '))) + CHAR(46) +
CONVERT(nvarchar(3),(CONVERT(int,' + @Hex3 + '))) + CHAR(46) +
CONVERT(nvarchar(3),(CONVERT(int,' + @Hex4 + ')))'
Set @Param = N'@Result nvarchar(16) output'
Exec master.dbo.Sp_executesql @SQL, @Param, @IP Output
END
```

---

**CODE SAMPLE 3.4—cont’d**

If we test the output, we get the results shown in Figure 3.15.

**FIGURE 3.15** Output of the New sp_UID2IP Stored Procedure
that SQL will validate the parameter format when the variable is passed into the procedure. If we try to pass non-GUID data, such as “Greg is a criminal” instead of ADA0C39B-FFFF-0000-0000-000000000000, then SQL generates the error shown in Figure 3.16.

This may seem obvious, but the procedure would work just fine if we had declared the input variable as nvarchar data instead of unique-identifier data:

```sql
CREATE PROCEDURE [dbo].[sp_UID2IP]
    @UIDLong uniqueidentifier
AS
BEGIN...
```

This could very well have been:

```sql
CREATE PROCEDURE [dbo].[sp_UID2IP]
    @UIDLong nvarchar(50)
AS
BEGIN...
```

And the process would have worked exactly the same from a conversion standpoint. However, Figure 3.17 shows what happens if we provide non-GUID data.

This is a very different error, and a much more serious one. You will see that the error occurred when SQL tried to figure out how to execute `Greg`, which shows that we can pass command data into the procedure. And while this particular procedure may obviate any actual SQL injection vulnerabilities by merit of its operational syntax, the fact remains that building strings within a process that are executed by way of the Sp_executesql system procedure can be dangerous. Some development policies explicitly prohibit the use of Sp_executesql in production environments for this very reason. It is actually a common limitation placed on developers in large or tightly controlled development environments.
An Editorial Note

The fact remains that developers use Sp_executesql all the time because the operational environments and functionality requirements they work with dictate that they do so. Developers use the tools they have to perform the functions they are given to design; that is their job. And while we want all developers to know about security risks and threats, we cannot expect them to be risk analysis experts. They are code functionality experts. Development and SDL policies are in place so security and risk analysts can look at the overall risk in allowing a particular development solution to be used in production and arrive at whatever guideline decisions they find appropriate. I agree with this. What I do not agree with is telling developers that they cannot use a particular function (i.e., Sp_executesql) while failing to provide them with functional equivalents that they can use.

Allow me to linger on this topic for a moment longer because I think this might help toward developing effective policy as opposed to just policy with this example in mind. Again, while you cannot expect developers to be security and risk analysis experts, you should ensure that they are aware of what problem you are trying to solve when you implement policy restrictions. It is important to have insight into the psychological economy of a developer in this process. If the developer does not explicitly know how SQL injection occurs, then just prohibiting Sp_executesql is not necessarily going to prevent it.

For instance, suppose I have to use a piece of system-generated data as part of a query—in this case, a piece of the GUID that I have to extract and convert to something else. As a SQL developer, it is my job to create T-SQL strings to get things done, so I may want to build a string and use Sp_executesql. However, the security guys tell me I cannot because of some injection issue, but they do not tell me what to do instead. So rather than just building the string, I retrieve the values, build them as I need them, and write...
them to a temp table. I then write a stored procedure to take parameterized character input to perform my function based on the data in the table. As far as meeting the code requirements are concerned, I have done my job because I have not used Sp_executesql, and I am happy because I did something creative that works well. My developer psychological economies of scale are balanced. However, the exact same SQL injection vector still exists because all an attacker has to do is alter the data in the table rather than the data in the string to accomplish the same attack. The mini-lesson here is that if you are going to limit what developers can and cannot do in code for security policies, make sure there are appropriate alternative solutions because it is in our nature to find a way to do what we want.

**Introducing SQL CLR**

We shall explore one of those alternative solutions now, and once done we can finally move on to creating the TMG Computer Sets we need to finish up. T-SQL is obviously a very powerful language, but it does have its limits. I believe that the previous data conversion and parsing examples illustrated that. While SQL is all about data management, sometimes T-SQL is not the most efficient way of dealing with highly customized data manipulation functions.

This is where we shall call upon SQL CLR as we have in other chapters. We will explore how we can use SQL CLR in C# code to accomplish the same thing we have done in T-SQL but without having to worry about SQL injection or other procedure- or function-based issues.

I will again reference you to read Chapter 1 in regard to the database configuration requirements necessary to post custom CLR assemblies into a SQL install and continue with the assumption that this has been done.

In my opinion, this is yet another perfect opportunity to familiarize you with the power of CLR, particularly in regard to how it supports your security posture and SDL requirements. Reviewing the fn_SQLString2Long function where we convert a dotted IP to an integer, we see the T-SQL implementation is as shown in Code Sample 3.5.

```
SET ANSI_NULLS ON
GO
SET QUOTED_IDENTIFIER ON
GO
-- ----------------------------------------------------------
-- Author:    Timothy "Thor" Mullen
-- Description:  Convert IP dot notation to integer
-- ----------------------------------------------------------
CREATE FUNCTION [dbo].[fn_SQLString2Long]
(

```

■ **CODE SAMPLE 3.5** fn_SQLString2Long: Reference Code Sample 3.2

(continued)
Using C# and CLR, we accomplish the same function with the simple function declaration shown in Code Sample 3.6.

```csharp
public static SqlInt64 fn_String2Long(string IP)
{
    SqlInt64 intAddress = BitConverter.ToUInt32(IPAddress.Parse(IP).GetAddressBytes().Reverse().ToArray(), 0);
    return intAddress;
}
```

**CODE SAMPLE 3.6** fn_String2Long: C# CLR to Convert Dotted IP to Integer (Int32)
Simple, eh? When this CLR function is published to our SQL server, it can then be called in the same way any other function is called, as we see when executed side to side, as shown in Figure 3.18.

Solving for the bulky and SQL injection susceptible sp_UID2IP procedure we wrote to convert the IP GUID into a string, we can meet our simplicity, security, and compliance goals with the CLR function definition shown in Code Sample 3.7.

```
public static SqlString fn_UID2String(string UID)
{
    string hexValue = UID.Substring(0, 8);
    UInt32 intAddress = UInt32.Parse(hexValue, System.Globalization.NumberStyles.HexNumber);

    IPAddress oldIP = new IPAddress(intAddress);
    byte[] me = oldIP.GetAddressBytes();
    Array.Reverse(me);
    UInt32 newAddress = BitConverter.ToUInt32(me, 0);
    IPAddress newIP = new IPAddress(newAddress);
    string IPString = newIP.ToString();
    return IPString;
}
```

In fact, we can securely build all of the necessary functions we have, as well as some new ones we will need, in less lines of code than the single function we found on the Internet. Code Sample 3.8 shows the entire bit of source code you need to post these CLR functions to your SQL server.

```
using System;
using System.Data;
using System.Data.SqlClient;
using System.Data.SqlTypes;
```

(continued)
using Microsoft.SqlServer.Server;
using System.Net;
using System.Linq;

public partial class UserDefinedFunctions
{
    public static SqlInt64 fn_String2Long(string IP)
    {
        SqlInt64 intAddress = BitConverter.ToUInt32(IPAddress.Parse(IP).
            GetAddressBytes().Reverse().ToArray(), 0);
        return intAddress;
    }

    public static SqlInt64 fn_UID2Long(string UID)
    {
        string hexValue = UID.Substring(0, 8);
        UInt32 intAddress = UInt32.Parse(hexValue, System.
            Globalization.NumberStyles.HexNumber);
        return intAddress;
    }

    public static SqlString fn_UID2String(string UID)
    {
        string hexValue = UID.Substring(0, 8);
        UInt32 intAddress = UInt32.Parse(hexValue, System.
            Globalization.NumberStyles.HexNumber);
        IPAddress oldIP = new IPAddress(intAddress);
        byte[] me = oldIP.GetAddressBytes();
        Array.Reverse(me);
        UInt32 newAddress = BitConverter.ToUInt32(me, 0);
        IPAddress newIP = new IPAddress(newAddress);
        string IPString = newIP.ToString();
        return IPString;
    }

    public static SqlString fn_Long2String(Int64 intAddress)
    {

        CODE SAMPLE 3.8—cont'd

Integrating with TMG 123
This collection of functions allows you to convert a dotted IP to an integer, a TMG GUID to an integer or dotted IP, and an integer to a dotted IP or GUID. In combination, you can perform all associated functions.

You should now have all the information and tools you need to report on any aspect of country-by-country traffic patterns. Now we are ready to build the sets in order for you to block or allow any or all traffic to or from anywhere you deem appropriate.

**Building ISA Server/TMG Computer Sets**

A Computer Set in ISA Server/TMG allows you to build a collection of differently named IP address ranges, individual computers, or subnets (in any combination) into a single object for use in access or publishing rules. This is the perfect object type to use in representing a country as each component range of the set is independent, but they can all be bound together as one object, as shown in Figure 3.19.

Computer Sets are very flexible objects, but they are difficult to work with because you must manually create each named entry one at a time in the UI. With more than 106,000 ranges in our data table, manual entry is not an
option. So we will have to programmatically create each country’s Computer Set container and then create each named range within that container.

If we refer to MSDN⁶ as a resource for accessing TMG objects from code, we will see Visual Basic for Applications (VBA) and C++ references. Knowing that we will need to access both code objects and data objects, I decided to use Microsoft Access as the interface for building our sets. This does a couple of things for us:

1. It gives us very easy access to the SQL data we already have for the IpToCountry data, and it fully supports VBA code.

2. The only way to make these sets portable is to export them from TMG as XML. We can also do that programmatically, but different versions of ISA Server (2004 and 2006, both Standard and Enterprise) have different XML tags by version and edition. ISA Server 2006 Standard Edition exports can also be imported into TMG, which makes things a bit easier, but to support multiple platforms, you still have to run the generation code from each server install. Doing this in Access is very easy because the VBA code stays the same even though the exported XML format is different.

For instance, here is an XML snippet for Papua New Guinea in ISA Server 2004:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<fpc4:Root xmlns:fpc4="http://schemas.microsoft.com/isa/
  -- config-4" xmlns:dt="urn:schemas-microsoft-com:
  -- datatypes"
  -- StorageName="FPC" StorageType="0">
  <fpc4:Build dt="string">4.0.2161.50</fpc4:Build>
  <fpc4:Comment dt="string">0</fpc4:Comment>
  <fpc4:Edition dt="int">80</fpc4:Edition>
  <fpc4:ExportItemClassCLSID dt="string">{0B964D61-5EBE-4837-9AE1-00FEF4ABC0B0}</fpc4:
    -- ExportItemClassCLSID>
    <fpc4:ExportItemStorageName dt="string">{20FBE63A-7146-442E-B695-F5C9E2FE65E5}</fpc4:
      -- ExportItemStorageName>
      <fpc4:IsaXmlVersion dt="string">1.0</fpc4:
        -- IsaXmlVersion>
        <fpc4:OptionalData dt="int">0</fpc4:OptionalData>
        <fpc4:Upgrade dt="boolean">0</fpc4:Upgrade>
        <fpc4:Arrays StorageName="Arrays" StorageType="0">
          <fpc4:Array StorageName="{BD3510F7-B661-4381-8898-3B31BE6EAB42}" StorageType="0">
            <fpc4:Components dt="int">-1</fpc4:Components>
            <fpc4:RuleElements StorageName="RuleElements"
              -- StorageType="0">
              <fpc4:ComputerSets StorageName="ComputerSets"
                -- StorageType="0">
                <fpc4:ComputerSet StorageName="{20FBE63A-7146-442E-B695-F5C9E2FE65E5}" StorageType="1">
                  <fpc4:Name dt="string">ThorSet_Papua New Guinea</fpc4:Name>
                </fpc4:ComputerSet>
              </fpc4:ComputerSets>
            </fpc4:RuleElements>
          </fpc4:Array>
        </fpc4:Arrays>
      </fpc4:Exports>
    </fpc4:Root>
```
And here is the same XML section in ISA Server 2006:

```xml
<?xml version="1.0" encoding="UTF-8"?>
<fpc4:Root xmlns:fpc4="http://schemas.microsoft.com/isa/
  -- config-4"
  xmlns:dt="urn:schemas-microsoft-com:datatypes"
  -- StorageName="FPC" StorageType="0">
  <fpc4:Build dt="string">5.0.5720.100</fpc4:Build>
  <fpc4:Comment dt="string">0</fpc4:Comment>
  <fpc4:Edition dt="int">16</fpc4:Edition>
  <fpc4:ExportItemClassCLSID dt="string">{0B964D61-
    -- 5EBE-4837-9AE1-00FEF4ABCB0F}</fpc4:
    -- ExportItemClassCLSID>
  <fpc4:ExportItemScope dt="int">0</fpc4:
    -- ExportItemScope>
  <fpc4:ExportItemStorageName dt="string">{9F5C68F1-
    -- 7E90-4A78-80E1-9390B4757596}</fpc4:
    -- ExportItemStorageName>
  <fpc4:IsaXmlVersion dt="string">5.30</fpc4:
    -- IsaXmlVersion>
  <fpc4:OptionalData dt="int">0</fpc4:OptionalData>
  <fpc4:Upgrade dt="boolean">0</fpc4:Upgrade>
  <fpc4:Arrays StorageName="Arrays" StorageType="0">
    <fpc4:Array StorageName="Array" StorageType="0">
      <fpc4:Components dt="int">1</fpc4:Components>
      <fpc4:DNSName dt="string"/>
      <fpc4:Name dt="string"/>
      <fpc4:RuleElements StorageName="RuleElements"
        -- StorageType="0">
        <fpc4:ComputerSets StorageName="ComputerSets"
          -- StorageType="0">
          <fpc4:ComputerSet StorageName="{9F5C68F1-7E90-
            -- 4A78-80E1-9390B4757596}" StorageType="1">
            <fpc4:Name dt="string">ThorSet_Papua New
              -- Guinea</fpc4:Name>
          </fpc4:ComputerSet>
          <fpc4:ComputerSet StorageName="{C4B4D104-4750-
            -- 9980-80E1-9390B4757596}" StorageType="1">
            <fpc4:Name dt="string">ThorSet_Papua New
              -- Guinea</fpc4:Name>
          </fpc4:ComputerSet>
        </fpc4:ComputerSets>
      </fpc4:RuleElements>
    </fpc4:Array>
  </fpc4:Arrays>
</fpc4:Root>
```

And here is the same XML section in ISA Server 2006:
Being able to easily export each different format style was important to me, so I needed something portable, stand-alone, and that supported data access and VBA. Now that TMG is the standard and ISA Server versions are no longer being developed, I will concentrate on writing up a C# version that is a bit faster and easier. But for now, we will have to use Access (or whatever else you would like) to get the job done quickly.

I will include the skeleton Access file as a resource for the book, but what I have done is created a linked reference in Access to my IpToCountry table in SQL, as well as a form with a couple buttons to which I have attached code, and a status window, as shown in Figure 3.20.

In design mode, I have Code Sample 3.9 built for the Create Sets (Write that Funky Stuff) button.

This code is written specifically for usability and flexibility during the TMG object creation process for execution directly by the user. SQL strings are dynamically created, and are not intended for accepting input from any external source, as it provides SQL injection capabilities.

Private Sub CreateSets_Click()
On Error Resume Next
'Grab Source IP Country data from SQL and Create TMG sets.
'
'Create the root object.
Dim root 'The FPCLib.FPC root object
Set root = CreateObject("FPC.Root")

CODE SAMPLE 3.9 Create Sets VBA Code to Create TMG Computer Set Objects
Declare the other objects needed.
Dim isaArray
Dim ComputerSets
Dim ComputerSet
Dim AddressRanges
Dim AddressRange
Dim rstCountries As Recordset
Dim rstAddresses As Recordset
Dim sCountry As String
Dim sSQL As String
```
Dim sRangeName As String
Dim sLogText As String

'Connect to array
Set isaArray = root.GetContainingArray()
Set ComputerSets = isaArray.RuleElements.ComputerSets

Log.SetFocus

'Get a distinct list of countries
'If this code somehow makes it to production, use parameterized queries.
sSQL = “SELECT distinct GeoIPCountry.CountryName FROM GeoIPCountry order by CountryName”

'Test sql to only get selected countries
'sSQL = “SELECT distinct GeoIPCountry.CountryName FROM GeoIPCountry where CountryName like ‘F*’ order by CountryName”

Set rstCountries = CurrentDb.OpenRecordset(sSQL)

Do 'Countries loop
    Log.Text = ""
sCountry = rstCountries!CountryName
    Set ComputerSet = ComputerSets.Add(“ThorSet_” + sCountry)
sSQL = “Select BegIP,EndIP,BegIPNo,EndIPNo,Country, CountryName from GeoIPCountry where CountryName = ’” + sCountry + “’ Order by BegIPNo”

    Set rstAddresses = CurrentDb.OpenRecordset(sSQL)
    Log.Text = Log.Text + Str(rstAddresses.RecordCount) + “address ranges found” + Constants.vbNewLine
    Do 'Addresses Loop
        sRangeName = Trim(rstAddresses!Country) +
        Trim(Str(rstAddresses!BegIPNo)) + “-“ + Trim(Str(rstAddresses!EndIPNo))
        Set AddressRanges = ComputerSet.AddressRanges
        Set AddressRange = AddressRanges.Add(sRangeName, rstAddresses!BegIP, rstAddresses!EndIP)
        rstAddresses.MoveNext
    Loop Until rstAddresses.EOF
    Log.Text = Log.Text + “...saving”
    'ComputerSet.Save
```

---

CODE SAMPLE 3.9—cont’d
This code creates the necessary TMG objects, creates a data record set representative of each distinct country, and then creates another record set of each country’s IP ranges. It moves through each top country record set creating the set, and moves through each range record for each country until all sets are created. At that point, it tells TMG to save the data via its save method. This will take some time, so be patient.

To export the data, the code shown in Code Sample 3.10 is bound to the ExportXML button.

```
Private Sub ExportSets_Click()
    ' On Error Resume Next
    ' Create the root object.
    Dim root ' The FPCLib.FPC root object
    Set root = CreateObject("FPC.Root")
    ' Declare the other objects needed.
    Dim isaArray
    Dim ComputerSets
    Dim ComputerSet
    Dim AddressRanges
    Dim AddressRange
    Dim sFilename As String
    Dim i As Integer

    Log.SetFocus
    Log.Text = ""
    ' Connect to array
    Set isaArray = root.GetContainingArray()
    Set ComputerSets = isaArray.RuleElements.ComputerSets
    i = 0
    For Each ComputerSet In ComputerSets
        i = i + 1
        ' Set ComputerSet = ComputerSets.Item
```

CODE SAMPLE 3.9—cont’d

CODE SAMPLE 3.10  Export Sets VBA Code to Export TMG Computer Set Objects to XML

(continued)
This code is much simpler and simply instantiates a Computer Set object, loops through each set present, and then exports it.

With the XML files, you can now select which countries you wish to work with and import them into whatever server you wish to. Once you have the sets, you can create whatever rules you wish to based on these sets. Figure 3.21 shows my SMTP inbound rule, which only allows SMTP from a select number of countries. I find that easier than blocking lots of other countries instead.

And with that, we wrap up this process example.
SUMMARY

In this chapter, we explored the methods of how you can research and analyze traffic from multiple different geolocations, and how to report on that data. We further illustrated different ways to block or allow data based on our wants and needs, learned how to manage our data within SQL server, and discovered how to use TMG programming objects to build Computer Sets to report from. We also introduced SQL CLR to handle some of the heavy lifting of data conversion.

REFERENCES


Belarus, (2011). FREE IP to country database (IPV4 and IPV6). In webnet77 Low cost domain names, domain transfers, web hosting, email accounts, and so much more, April 22, http://software77.net/geo-ip/.
