5

AUTOMATING NESSUS

Nessus is a popular and powerful *vulnerability scanner* that uses a database of known vulnerabilities to assess whether a given system on a network is missing any patches or is vulnerable to known exploits. In this chapter, I'll show you how to write classes to interact with the Nessus API to automate, configure, and run a vulnerability scan.

Nessus was first developed as an open source vulnerability scanner, but it became closed source in 2005 after being purchased by Tenable Network Security. As of this writing, Tenable offers a seven-day trial of Nessus Professional and a limited version called Nessus Home. The biggest difference between the two is that Nessus Home allows you to scan only 16 IP addresses at once, but Home should be sufficient for you to run the examples in this chapter and become familiar with the program. Nessus is particularly popular with professionals who help scan and manage other companies' networks. Follow the instructions on the Tenable site *https://www.tenable.com/products/nessus-home/* to install and configure Nessus Home.

Many organizations require regular vulnerability and patch scanning in order to manage and identify risks on their network, as well as for compliance purposes. We'll use Nessus to accomplish these goals by building classes to help us perform unauthenticated vulnerability scans against hosts on a network.

REST and the Nessus API

The advent of web applications and APIs has given rise to an architecture of APIs called REST APIs. *REST (representational state transfer)* is a way of accessing and interacting with resources (such as user accounts or vulnerability scans) on the server, usually over HTTP, using a variety of HTTP methods (GET, POST, DELETE, and PUT). HTTP methods describe our intent in making the HTTP request (for example, do we want to create a resource or modify a resource?), kind of like CRUD (Create, Read, Update, Delete) operations in databases.

For instance, take a look at the following simple GET HTTP request, which is like a read operation for a database (like `SELECT * FROM users WHERE id = 1`):

```
GET /users/❶1 HTTP/1.0
Host: 192.168.0.11
```

In this example, we're requesting information for the user with an ID of 1. To get the information for another user's ID, you could replace the 1 ❶ at the end of the URI with that user's ID.

To update the information for the first user, the HTTP request might look like this:

```
POST /users/1 HTTP/1.0
Host: 192.168.0.11
Content-Type: application/json
Content-Length: 24

{"name": "Brandon Perry"}
```

In our hypothetical RESTful API, the preceding POST request would update the first user's name to Brandon Perry. Commonly, POST requests are used to update a resource on the web server.

To delete the account entirely, use DELETE, like so:

```
DELETE /users/1 HTTP/1.0
Host: 192.168.0.11
```

The Nessus API will behave similarly. When consuming the API, we'll send JSON to and receive JSON from the server, as in these examples. The classes we'll write in this chapter are designed to handle the ways that we communicate and interact with the REST API.

Once you have Nessus installed, you can find the Nessus REST API documentation at *https://<IP address>:8834/api*. We'll cover only a few of the core API calls used to drive Nessus to perform vulnerability scans.

The NessusSession Class

To automate sending commands and receiving responses from Nessus, we'll create a session with the `NessusSession` class and execute API commands, as shown in Listing 5-1.

```
public class NessusSession : ❶IDisposable
{
  public ❷NessusSession(string host, string username, string password)
  {

    ServicePointManager.ServerCertificateValidationCallback =
      (Object obj, X509Certificate certificate, X509Chain chain, SslPolicyErrors errors) => true;

    this.Host = ❸host;

    if (❹!Authenticate(username, password))
      throw new Exception("Authentication failed");
  }

  public bool ❺Authenticate(string username, string password)
  {
    JObject obj = ❻new JObject();
    obj["username"] = username;
    obj["password"] = password;

    JObject ret = ❼MakeRequest(WebRequestMethods.Http.Post, "/session", obj);

    if (ret ["token"] == null)
      return false;

    this.❽Token = ret["token"].Value<string>();
    this.Authenticated = true;

    return true;
  }
```

Listing 5-1: The beginning of the `NessusSession` class showing the constructor and `Authenticate()` method

As you can see in Listing 5-1, this class implements the `IDisposable` interface ❶ so that we can use the `NessusSession` class within a `using` statement. As you may recall from earlier chapters, the `IDisposable` interface allows us to automatically clean up our session with Nessus by calling `Dispose()`, which we'll implement shortly, when the currently instantiated class in the `using` statement is disposed during garbage collection.

At ❸, we assign the `Host` property to the value of the `host` parameter passed to the `NessusSession` constructor ❷, and then we try to authenticate ❹ since any subsequent API calls will require an authenticated session. If authentication fails, we throw an exception and print the alert "Authentication failed". If authentication succeeds, we store the API key for later use.

In the Authenticate() method ❺, we create a JObject ❻ to hold the credentials passed in as arguments. We'll use these to attempt to authenticate, and then we'll call the MakeRequest() method ❼ (discussed next) and pass the HTTP method, the URI of the target host, and the JObject. If authentication succeeds, MakeRequest() should return a JObject with an authentication token; if authentication fails, it should return an empty JObject.

When we receive the authentication token, we assign its value to the Token property ❽, assign the Authenticated property to true, and return true to the caller method to tell the programmer that authentication succeeded. If authentication fails, we return false.

Making the HTTP Requests

The MakeRequest() method makes the actual HTTP requests and then returns the responses, as shown in Listing 5-2.

```
public JObject MakeRequest(string method, string uri, ❶JObject data = null, string token = null)
{
  string url = ❷"https://" + this.Host + ":8834" + uri;
  HttpWebRequest request = (HttpWebRequest)WebRequest.Create(url);
  request.❸Method = method;

  if (!string.IsNullOrEmpty(token))
    request.Headers ["X-Cookie"] = ❹"token=" + token;

  request.❺ContentType = "application/json";

  if (data != null)
  {
    byte[] bytes = System.Text.Encoding.ASCII.❻GetBytes(data.ToString());
    request.ContentLength = bytes.Length;
    using (Stream requestStream = request.GetRequestStream())
      requestStream.❼Write(bytes, 0, bytes.Length);
  }
  else
    request.ContentLength = 0;

  string response = string.Empty;
  try ❽
  {
    using (StreamReader reader = new ❾StreamReader(request.GetResponse().GetResponseStream()))
    response = reader.ReadToEnd();
  }
  catch
  {
    return new JObject();
  }

  if (string.IsNullOrEmpty(response))
    return new JObject();
  return JObject.❿Parse(response);
}
```

Listing 5-2: The `MakeRequest()` method from the `NessusSession` class

The `MakeRequest()` method has two required parameters (HTTP and URI) and two optional ones (the `JObject` and the authentication token). The default value for each is null.

To create `MakeRequest()`, we first create the base URL for the API calls ❷ by combining the host and URI parameters and passing in the result as the second argument; then we use `HttpWebRequest` to build the HTTP request and set the property of `HttpWebRequest` `Method` ❸ to the value of the `method` variable passed into `MakeRequest()` method. Next, we test whether the user supplied an authentication token in `JObject`. If so, we assign the HTTP request header `X-Cookie` to the value of the `token` parameter ❹, which Nessus will look for when we authenticate. We set the `ContentType` property ❺ of the HTTP request to `application/json` to ensure that the API server knows how to deal with the data we are sending in the body of the request (otherwise, it will refuse to accept the request).

If a `JObject` is passed to `MakeRequest()` in the third argument ❶, we convert it to a byte array using `GetBytes()` ❻, because the `Write()` method can only write bytes. We assign the `ContentLength` property to the size of the array and then use `Write()` ❼ to write the JSON to the request stream. If the `JObject` passed to `MakeRequest()` is null, we simply assign the value 0 to `ContentLength` and move on, since we will not be putting any data in the request body.

Having declared an empty string to hold the response from the server, we begin a `try`/`catch` block at ❽ to receive the response. Within a `using` statement, we create a `StreamReader` ❾ to read the HTTP response by passing the server's HTTP response stream to the `StreamReader` constructor; then we call `ReadToEnd()` to read the full response body into our empty string. If reading the response causes an exception, we can expect that the response body is empty, so we catch the exception and return an empty `JObject` to `ReadToEnd()`. Otherwise, we pass the response to `Parse()` ❿ and return the resulting `JObject`.

Logging Out and Cleaning Up

To finish the `NessusSession` class, we'll create `LogOut()` to log us out of the server and `Dispose()` to implement the `IDisposable` interface, as shown in Listing 5-3.

```
public void ❶LogOut()
{
  if (this.Authenticated)
  {
    MakeRequest("DELETE", "/session", null, this.Token);
    this.Authenticated = false;
  }
}
```

```
  public void ❷Dispose()
  {
    if (this.Authenticated)
      this.LogOut();
  }

  public string Host { get; set; }
  public bool Authenticated { get; private set; }
  public string Token { get; private set; }
}
```

Listing 5-3: The last two methods of the `NessusSession` *class, as well as the* `Host`*,* `Authenticated`*, and* `Token` *properties*

The `LogOut()` method ❶ tests whether we're authenticated with the Nessus server. If so, we call `MakeRequest()` by passing `DELETE` as the HTTP method; `/session` as the URI; and the authentication token, which sends a DELETE HTTP request to the Nessus server, effectively logging us out. Once the request is complete, we set the `Authenticated` property to `false`. In order to implement the `IDisposable` interface, we create `Dispose()` ❷ to log us out if we are authenticated.

Testing the NessusSession Class

We can easily test the `NessusSession` class with a small `Main()` method, as shown in Listing 5-4.

```
public static void ❶Main(string[] args)
{
 ❷using (NessusSession session = new ❸NessusSession("192.168.1.14", "admin", "password"))
  {
    Console.❹WriteLine("Your authentication token is: " + session.Token);
  }
}
```

Listing 5-4: Testing the `NessusSession` *class to authenticate with* `NessusManager`

In the `Main()` method ❶, we create a new `NessusSession` ❸ and pass the IP address of the Nessus host, the username, and the Nessus password as the arguments. With the authenticated session, we print the authentication token ❹ Nessus gave us on successful authentication and then exit.

NOTE *The* `NessusSession` *is created in the context of a* `using` *statement* ❷*, so the* `Dispose()` *method we implemented in the* `NessusSession` *class will be automatically called when the* `using` *block ends. This logs out the* `NessusSession`*, invalidating the authentication token we were given by Nessus.*

Running this code should print an authentication token similar to the one in Listing 5-5.

```
$ mono ./ch5_automating_nessus.exe
Your authentication token is: 19daad2f2fca99b2a2d48febb2424966a99727c19252966a
$
```

Listing 5-5: Running the `NessusSession` test code to print the authentication token

The NessusManager Class

Listing 5-6 shows the methods we need to implement in the `NessusManager` class, which will wrap common API calls and functionality for Nessus in easy-to-use methods we can call later.

```
public class NessusManager : ❶IDisposable
{
  NessusSession _session;
  public NessusManager(NessusSession session)
  {
    _session = ❷session;
  }

  public JObject GetScanPolicies()
  {
    return _session.❸MakeRequest("GET", "/editor/policy/templates", null, _session.Token);
  }

  public JObject CreateScan(string policyID, string cidr, string name, string description)
  {
    JObject data = ❹new JObject();
    data["uuid"] = policyID;
    data["settings"] = new JObject();
    data["settings"]["name"] = name;
    data["settings"]["text_targets"] = cidr;
    data["settings"]["description"] = description;

    return _session.❺MakeRequest("POST", "/scans", data, _session.Token);
  }

  public JObject StartScan(int scanID)
  {
    return _session.MakeRequest("POST", "/scans/" + scanID + "/launch", null, _session.Token);
  }

  public JObject ❻GetScan(int scanID)
  {
    return _session.MakeRequest("GET", "/scans/" + scanID, null, _session.Token);
  }

  public void Dispose()
  {
    if (_session.Authenticated)
      _session.❼LogOut();
    _session = null;
  }
}
```

Listing 5-6: The `NessusManager` *class*

The `NessusManager` class implements `IDisposable` ❶ so that we can use `NessusSession` to interact with the Nessus API and log out automatically if necessary. The `NessusManager` constructor takes one argument, a `NessusSession`, and assigns it to the private `_session` variable ❷, which any method in `NessusManager` can access.

Nessus is preconfigured with a few different scan policies. We'll sort through these policies using `GetScanPolicies()` and `MakeRequest()` ❸ to retrieve a list of policies and their IDs from the */editor/policy/templates* URI. The first argument to `CreateScan()` is the scan policy ID, and the second is the CIDR range to scan. (You can also enter a newline-delimited string of IP addresses in this argument.)

The third and fourth arguments can be used to hold a name and description of the scan, respectively. We'll use a unique `Guid` (*globally unique ID*, long strings of unique letters and numbers) for each names since our scan is only for testing purposes, but as you build more sophisticated automation, you may want to adopt a system of naming scans in order to make them easier to track. We use the arguments passed to `CreateScan()` to create a new `JObject` ❹ containing the settings for the scan to create. We then pass this `JObject` to `MakeRequest()` ❺, which will send a POST request to the */scans* URI and return all relevant information about the particular scan, showing that we successfully created (but did not start!) a scan. We can use the scan ID to report the status of a scan.

Once we've created the scan with `CreateScan()`, we'll pass its ID to the `StartScan()` method, which will create a POST request to the */scans/<scanID>/launch* URI and return the JSON response telling us whether the scan was launched. We can use `GetScan()` ❻ to monitor the scan.

To complete `NessusManager`, we implement `Dispose()` to log out of the session ❼ and then clean up by setting the `_session` variable to null.

Performing a Nessus Scan

Listing 5-7 shows how to begin using `NessusSession` and `NessusManager` to run a scan and print the results.

```
public static void Main(string[] args)
{
  ServicePointManager.❶ServerCertificateValidationCallback =
    (Object obj, X509Certificate certificate, X509Chain chain, SslPolicyErrors errors) => true;

  using (NessusSession session = ❷new NessusSession("192.168.1.14", "admin", "password"))
  {
    using (NessusManager manager = new NessusManager(session))
    {
```

```
    JObject policies = manager.❸GetScanPolicies();
    string discoveryPolicyID = string.Empty;
    foreach (JObject template in policies["templates"])
    {
      if (template ["name"].Value<string>() == ❹"basic")
        discoveryPolicyID = template ["uuid"].Value<string>();
    }
```

Listing 5-7: Retrieving the list of scan policies so we can start a scan with the correct scan policy

We begin our automation by first disabling SSL certificate verification (because the Nessus server's SSL keys are self-signed, they will fail verification) by assigning an anonymous method that only returns `true` to the `ServerCertificateValidationCallback` ❶. This callback is used by the HTTP networking libraries to verify an SSL certificate. Simply returning `true` causes any SSL certificate to be accepted. Next, we create a `NessusSession` ❷ and pass it the IP address of the Nessus server as well as the username and password for the Nessus API. If authentication succeeds, we pass the new session to another `NessusManager`.

Once we have an authenticated session and a manager, we can begin interacting with the Nessus server. We first get a list of the scan policies available with `GetScanPolicies()` ❸ and then create an empty string with `string.Empty` to hold the scan policy ID for the basic scan policy and iterate over the scan policy templates. As we iterate over the scan policies, we check whether the name of the current scan policy equals the string `basic` ❹; this is a good starting point for a scan policy that allows us to perform a small set of unauthenticated checks against hosts on the network. We store the ID for the basic scan policy for later use.

Now to create and start the scan with the basic scan policy ID, as shown in Listing 5-8.

```
    JObject scan = manager.❶CreateScan(discoveryPolicyID, "192.168.1.31",
       "Network Scan", "A simple scan of a single IP address.");
    int scanID = ❷scan["scan"]["id"].Value<int>();
    manager.❸StartScan(scanID);
    JObject scanStatus = manager.GetScan(scanID);

    while (scanStatus["info"]["status"].Value<string>() != ❹"completed")
    {
      Console.WriteLine("Scan status: " + scanStatus["info"]
          ["status"].Value<string>());
      Thread.Sleep(5000);
      scanStatus = manager.❺GetScan(scanID);
    }

    foreach (JObject vuln in scanStatus["vulnerabilities"])
      Console.WriteLine(vuln.ToString());
  }
}
```

Listing 5-8: The second half of the Nessus automation `Main()` method

At ❶, we call `CreateScan()`, passing in a policy ID, IP address, name, and description of the method, and we store its response in a `JObject`. We then pull the scan ID out of the `JObject` ❷ so that we can pass the scan ID to `StartScan()` ❸ to start the scan.

We use `GetScan()` to monitor the scan by passing it the scan ID, storing the result in a `JObject` and using a `while` loop to continually check whether the current scan status has completed ❹. If the scan has not completed, we print its status, sleep for five seconds, and call `GetScan()` ❺ again. The loop repeats until the scan reports completed, at which point we iterate over and print each vulnerability returned by `GetScan()` in a `foreach` loop, which may look something like Listing 5-9. A scan might take several minutes to complete, depending on your computer and network speed.

```
$ mono ch5_automating_nessus.exe
Scan status: running
Scan status: running
Scan status: running
--snip--
{
  "count": 1,
  "plugin_name": ❶"SSL Version 2 and 3 Protocol Detection",
  "vuln_index": 62,
  "severity": 2,
  "plugin_id": 20007,
  "severity_index": 30,
  "plugin_family": "Service detection"
}
{
  "count": 1,
  "plugin_name": ❷"SSL Self-Signed Certificate",
  "vuln_index": 61,
  "severity": 2,
  "plugin_id": 57582,
  "severity_index": 31,
  "plugin_family": "General"
}
{
  "count": 1,
  "plugin_name": "SSL Certificate Cannot Be Trusted",
  "vuln_index": 56,
  "severity": 2,
  "plugin_id": 51192,
  "severity_index": 32,
  "plugin_family": "General"
}
```

Listing 5-9: Partial output from an automated scan using the Nessus vulnerability scanner

The scan results tell us that the target is using weak SSL modes (protocols 2 and 3) ❶ and a self-signed SSL certificate on an open port ❷. We can now ensure that the server's SSL configurations are using fully up-to-date

SSL modes and then disable the weak modes (or disable the service altogether). Once finished, we can rerun our automated scan to ensure that Nessus no longer reports any weak SSL modes in use.

Conclusion

This chapter has shown you how to automate various aspects of the Nessus API in order to complete an unauthenticated scan of a network-attached device. In order to achieve this, we needed to be able to send API requests to the Nessus HTTP server. To do so, we created the `NessusSession` class; then, once we were able to authenticate with Nessus, we created the `NessusManager` class to create, run, and report the results of a scan. We wrapped everything with code that used these classes to drive the Nessus API automatically based on user-provided information.

This isn't the extent of the features Nessus provides, and you'll find more detail in the Nessus API documentation. Many organizations require performing authenticated scans against hosts on the network in order to get full patch listings to determine host health, and upgrading our automation to handle this would be a good exercise.