Securing Network Communication with Stunnel, OpenSSH, and OpenVPN

Kurt Kincaid

Table of Contents

Abstract

Networks are increasingly complex, support a wide variety of applications and network services, and handle converged data, voice, and video traffic across both wired and wireless networks using numerous platforms. As a result of HIPAA, Sarbanes-Oxley, and similar regulations, in addition to recent high-profile stories involving the compromise of personal information, maintaining the confidentiality and integrity of data in transit is of paramount importance. This article covers securing network communication, including legacy systems, using such Open Source tools as Stunnel, OpenSSH, and OpenVPN.

Introduction

Among the more difficult tasks that must be confronted by IT shops is that of securing data in transit. What if you have an old, legacy application that has no support for encrypted communication? What if you don't have the time and/or money to upgrade that old system to the latest version which ***does*** support encrypted communication? Or what if you simply can't afford the downtime required for such an upgrade? Or what if your application has support for encrypted communication, but as the administrator, you decide that their implementation does not meet your minimum requirements? These are all issues that are addressed relatively easily with Stunnel, OpenSSH, and OpenVPN.

Who needs this?

At the risk of overstating, almost everyone. If you are sending data in clear text across your network, ask yourself this question: What happens if someone, right now, is sniffing every packet I send across my network? How badly am I exposed? In my experience, unless extreme security measures have already been taken, you might be surprised. Or shocked. Or perhaps even horrified.

What if I'm on a switched network?

First, let's clarify switched and shared networks. Older (and cheaper) networking gear, like hubs, is shared. Let's say we only have three nodes on our tiny network. On a shared network, traffic that is destined for node A is also sent to nodes B and C. These nodes, however, are smart enough to know that the traffic isn't addressed to them, so they silently ignore it. This is a dream come true for our Black Hat. All it takes is a simple packet sniffer like Ethereal or tcpdump or even Snort, and our Black Hat can listen to every single packet on our network. By contrast, we have switched networks. On a switched network, node A receives only traffic that is destined for itself, as well as broadcast and multicast traffic. Node A, under normal circumstances, can't see traffic destined for nodes B and C. This is an important step in securing your network. In general, there is no need for a node to be able to see network traffic that is destined for someone else. It also has the added benefit of substantially decreasing the amount of unnecessary network traffic. So if you're on a switched network, you're safe, right? Sadly, no. Tools like Ettercap make it all too easy to sniff traffic on a switched network. So while a switched network is a big step in the right direction, you still need additional weapons in your arsenal to protect yourself and your network from the would-be Black Hat.

How secure is this?

There are two key issues that have to be considered in order to adequately answer this question. First there is the matter of the level of encryption supported by the tools. All three of these tools support strong encryption. In that regard, these tools are quite secure. The second issue that has to be considered, however, is that of the code itself. In general, the code for these tools is fast and secure. However, it is the nature of things that vulnerabilities are discovered, even in otherwise good code. That being the case, if you intend to use these tools (or any other security tools, for that matter), it behooves you to stay on top of vulnerability advisories. If a vulnerability is discovered in the version of one of the tools you are using, patch it. Don't put it off. Doing so only invites trouble. Always plan for Murphy's Law. It is the nature of Murphy's Law that it strikes when you least expect it. Though frustrating, Murphy's Law is important because it keeps you honest. Plus, when something ***does*** go wrong, it presents an opportunity to demonstrate that you have planned for these contingencies. When it comes to network security, this is always a good trait.

Background

Before we roll up our sleeves and get our hands dirty, there are a few background items that should be mentioned.

1. One of the key points to keep in mind with all of the tools discussed herein is that they are all Open Source. This means that in terms of out-of-pocket expense, there is none. This can be an important issue when it comes to getting sign-off of upper management.

2. These tools are multi-platform. I have personally used these tools on Windows XP, Windows Server 2003, Windows 2000, Solaris, AIX, and numerous distributions of Linux. The one exception is OpenVPN. As of this writing, OpenVPN does not work on AIX.

3. When it comes to securing your data in transit, you have basically two options. You can encrypt the data and then send it over an otherwise unsecured channel, or you can create an encrypted tunnel and send the unencrypted data through the encrypted tunnel. All three of these tools employ the latter.

4. OpenSSL is the cryptographic library behind all three of these tools. As such, you have the full power of OpenSSL at your disposal.

Finally, there are a few caveats with OpenSSL and OpenSSH that must be addressed before going any further. Though these items are relatively easy to address, the specifics on how to do this fall outside the scope of this article and are left as an exercise for the reader. There is copious documentation readily available on the internet.

- OpenSSL

    1. Disallow SSL version 2, commonly seen as SSLv2. SSLv2 is vulnerable to the ciphersuite rollback attack. While this doesn't directly expose your data, it can cause the data to be encrypted with very low encryption, potentially allowing a Black Hat to decrypt your data at a later date.

    2. OpenSSL allows you to specify which encryption methods you want to use and which methods you don't want to use. Take advantage of this and expressly disallow any low level encryption methods.

- OpenSSH

    1. Disallow SSH version 1, commonly seen as SSHv1. Tools like Ettercap can decrypt SSHv1 traffic in real time. I've done this in a lab environment, so I can personally vouch for the ease with which this can be done.

    2. Increase the key size to as large as is feasible for your environment. Similarly, decrease the time between regenerating the key to as little as is feasible for your environment. Don't over do it, though. If you are generating huge keys too frequently, performance will take a tremendous hit.

Assessing the Need

Before determining which tool to use and how to go about deployment, one must first assess the need. In some situations, a simple Stunnel implementation will suffice. In other situations, something substantially more complex may be necessary.

- ***Scope***. First order of business is to determine the scope of the project. Do we need to secure a single service or many? How many clients? Are there any restrictions with regard to installing software on the client machines? What are the chances that this implementation will need to grow over time?
- ***Complexity tolerance***. This applies to both the users and to the administrators who will be managing the implementation. Are you a small IT shop? If so, you'll probably want to keep things as simple as possible. If you are a larger shop where duties can be more evenly distributed, you can generally afford to implement something a bit more complex. What are your users like? Are they relatively tech-savy? Or are they something less than savy?
- ***Level of available expertise***. Do you have a number of people who can handle the long-term administration? Or are you a one-person-show?

Be brutally honest with yourself on these points. The proverbial Kiss of Death is to be waist deep in your implementation before coming to the realization that you are using the wrong tool for the wrong job.

Stunnel

Of the three tools we'll be discussing, Stunnel is the simplest; it is lightweight, does one thing, and does it very well. In most cases, you'll want to use Stunnel when the communication requirements are simple, both in terms of complexity and quantity. For example, we might want to add SSL support to a web server that doesn't have SSL support. Let's say we want to run Apache on a Windows box. Apache is readily available for Windows, but Apache with SSL for Windows is a bit more problematic. We could get a third party distribution like XAMPP, but in some circumstances, that might be overkill. We could try to compile Apache with mod_ssl for ourselves. This may sound relatively simple, but trust me, it isn't. Stunnel to the rescue. With Stunnel, we can add SSL support in, literally, a matter of seconds.

Another case where we might want to use Stunnel is when we can't or don't want to establish and maintain a tunnel between the client and server. Let's say we have users who are telecommuting and they need to access a server on the inside of the enterprise firewall. Good firewall practices says we should set the connection timeout very low. If we establish a tunnel and due to lack of activity the firewall drops the connection, users are inconvenienced by having to login again. Stunnel is stateless, so we can conveniently work around this issue.

We may also decide to use Stunnel when we don't mind opening up additional ports on the server for each service we want to secure. It is important to note that this is a one-to-one ratio. For each port we want to secure, we must open another port for Stunnel.

Adding SSL to a Web Server

Let's continue with the example mentioned previously: adding SSL support to a web server. In this particular case, since web browsers already know how to speak SSL, there is absolutely no client-side setup. On the server side, we setup Stunnel to listen for remote TCP requests on port 443 (this is the standard port for HTTPS, encrypted HTTP traffic) and to redirect locally to port 80, the standard HTTP port.

Example 1. Server Side Config

```
cert    = /some/directory/server.crt
key     = /some/directory/server.key
ciphers = TLSv1:SSLv3:!SSLv2:!LOW:@STRENGTH

[https]
accept  = 443
connect = 80
```

Going through the above config file, the first two lines are specifying the certificate and key to be used for the server. These can be created using OpenSSL. Creating keys and signing certificates can be rather complicated and falls well outside our scope. However, to ease this process, I use another Open Source tool called XCA to handle all of the complicated stuff. There is plenty of documentation available on creating certificates and keys by hand, but personally, I wouldn't dream of doing it without XCA. The third line specifies which ciphers to use. We are explicitly saying we want to use TLSv1 (Transport Layer Security) and SSLv3. In addition, we disallow SSLv2 and low encryption ciphers. The particularly interesting item here is the last entry, @STRENGTH, which says to sort the available ciphers by strength, from highest to lowest. This means that Stunnel will try to establish a connection using its highest possible level of encryption first. The last section says that we are going to accept connections on port 443 and connect locally to port 80. With our config file in place, we start our stunnel instance like this:

```
# stunnel /path/to/stunnel.conf
```

Encrypting Telnet with Stunnel

Now we'll move on to a slightly more complicated example. This time we're going to encrypt telnet over stunnel. Conceptually, we start Stunnel on the client listening on arbitrary port X and forwarding to arbitrary port Y on our telnet server. On the server, we have another instance of Stunnel running that is listening for remote requests on arbitrary port Y and forwarding locally to port 23, the telnet port.

Example 2. Client Side Config

```
cert    = /my/directory/my.crt
key     = /my/directory/my.key
client  = yes
ciphers = TLSv1:SSLv3:!SSLv2:!LOW:@STRENGTH

[my_label]
accept  = 25000
connect = my.telnet.server:25000
```

The config is very similar to our last example. We specify our client certificate and key, then we explicitly state that we are a client. We then specify our ciphers, state that we are going to accept connections on local port 25000 (this can be any unused port), and we are redirecting to our telnet server on port 25000. Again, this can be any unused port on the server. There is no requirement stating that they have to be the same, though I try to make them the same on client on server whenver I can just for tidiness reasons. Now let's look at the server config.

Example 3. Server Side Config

```
cert    = /some/directory/server.crt
key     = /some/directory/server.key
ciphers = TLSv1:SSLv3:!SSLv2:!LOW:@STRENGTH

[my_label]
accept  = 25000
connect = 23
```

Again, this config is very similar to the other config files we've used. We specify certificate and key for our server, then we specify the ciphers we want to use. Finally, we state that we're accepting connections on port 25000 and redirecting those requests locally to port 23. On both client and server, Stunnel is started in exactly the same way as in Example 1. Now to initiate our encrypted telnet session, we do the following:

```
# telnet localhost 25000
```

Stunnel Conclusions

- Pros
    - Stunnel is easy to set up. Most setups, quite literally, only take a minute or two.
    - Stunnel is secure. You can use encryption as high as OpenSSL supports.
    - Multiple tunnels can be setup in a single config file, thus requiring only a single instance of stunnel on the server.
    - As long as we're going to use an non-privileged port, Stunnel can be run by a non-privileged user.
- Cons
    - A separate tunnel is required for each service to be secured.
    - Stunnel requires a modest understanding of setting up and using Certificate Authorities. As mentioned previously, though, tools like XCA can handle most of this for you.
    - Stunnel bleeds the approximate length of the transaction. So what does that mean? Since Stunnel doesn't maintain a constant tunnel with the server, a potential Black Hat can see exactly when your

communication with the server begins and ends. The content is encrypted, so you're safe as far as that is concerned. However, a skilled Black Hat can make some fairly accurate assumptions based on how much content is sent, when, and how often. Understand that this sort of reconnaissance is difficult, and if you're dealing with someone good enough to tackle things at this level, you likely have far greater problems on your hands. It is, however, something that must be taken into consideration.

There are a few final points about Stunnel that one must keep in mind. Neither pros, nor cons, these are simply things that you need to be aware of prior to deploying Stunnel. Stunnel does not require user-level authentication at the server level. This can be a good thing if you want to minimize the inconvenience for your users. Stunnel does not compress the data sent through the tunnel. That means a particularly skilled Black Hat has more opportunity to perform cryptanalysis on the data. As mentioned before, though, if you're dealing with a Black Hat skilled enough to be doing this sort of thing, you need to sound a Red Alert and call all hands to battlestations...you've got a major issue on your hands.

OpenSSH

SSH is the ***de facto*** replacement for telnet, and OpenSSH is one of the most popular implementations thereof. In the simplest of terms, SSH is an encrypted version of telnet. SSH, however, can do things you wouldn't dream of doing with telnet. For our purposes, we're interested in tunneling. That being said, when would we want to use OpenSSH for tunneling over Stunnel? If you already have OpenSSH installed and running, then this is the solution for you. OpenSSH is a standard part of most Linux distributions and is increasingly common with many UNIX distributions as well. If it isn't part of the standard install, chances are it is still readily available. There are also several ports of OpenSSH to the Windows platform, which is an added bonus if you're in a Windows-centered environment. In addition, you would also chose OpenSSH for tunneling if you have multiple services you want to secure and you don't want to open up an additional port on the server for each, as you would have to do with Stunnel.

SSH tunneling is accomplished by port forwarding through an established SSH connection. The user on the client side initiates a specially crafted SSH connection to the server, specifying the listening port on the client side and the ultimate destination port on the server side. From the server's perspective, assuming we already have OpenSSH installed and running, no additional configuration is necessary.

Encrypting telnet with OpenSSH

Example 4. Tunneling telnet

```
# ssh -C -l my_username 5000:localhost:23 my.telnet.server
```

Let's step through the above example. First we enable compression with the "-C" flag, then we specify the username and the port we're going to listen on locally, in this case, port 5000. We are then ultimately redirecting to port 23 on my.telnet.server. Once we've established this connection, we can start a telnet session tunneled through our SSH connection like so: **`telnet localhost 5000`**

Encrypting MySQL with OpenSSH

Example 5. Tunneling MySQL

```
# ssh -C -l username 5001:localhost:3306 my.mysql.server
```

Establishing the SSH connection is almost exactly the same as it was in the previous example. The only differences are that we chose to use local port 5001 and we're ultimately redirecting to port 3306 on my.mysql.server. Once done, we initiate our MySQL session like so:

```
mysql --host=localhost --port=5001 --username=username --password=pwd
```

OpenSSH Conclusions

- Pros
    - Assuming OpenSSH is already installed and running on our server, no special setup is required on the server side. The potential exception is if you have disabled forwarding in your OpenSSH config. The default is to have forwarding enabled, so unless you have specifically disabled this feature, you should be ready for action.
    - The persistent tunnel, plus the addition of compression, partially masks the transaction length.
    - The OpenSSH log provides detailed audit trail of tunnel usage.
    - Multiple services can be secured without opening ports on the server other than port 22, the standard SSH port.
    - As long as we're using a non-privileged port, the OpenSSH tunnel can be setup by a non-privileged user.
- Cons
    - Users are required to authenticate at the server level in order to establish the OpenSSH tunnel. If the service we're trying to secure ***also*** requires authentication, the user will have to authenticate a second time. In turn, this means that there will be twice as many logins on the server, two per user. Neither of these items are necessarily a problem, but they are things you'll have to be aware of. You can use key authentication in OpenSSH to expedite the tunnel login, and the number of logins on the server will really only matter if you have license issues with your OS that restrict the number of simultaneous logins.
    - Encryption ciphers are specified at the client-level, not the server. Again, not necessarily a problem, but it does mean that the user could potentially select an encryption algorithm that might not meet your minimum standards.

Some final thoughts on OpenSSH tunneling. First, you should probably never use the "-g" option when setting up the tunnel. This enables the gateway option and makes it so someone else could use your tunnel to the server. This should generally be avoided because it doesn't provided end-to-end encryption for the person using your tunnel and it obfuscates the audit trail. Second, you should probably never use host-based authentication with OpenSSH. Host-based authentication allows you to say that Server X is allowed in without further authentication. This can be a problem if our would-be Black Hat spoofs Server X's MAC address and knocks Server X offline. If we were using host-based authentication, our Black Hat would now have unfettered access.

OpenVPN

OpenVPN is a tool that has a lot of untapped potential. SSL VPNs have been a hot topic lately, and OpenVPN is, in my opinion, one of the best. OpenVPN has two modes of operation. It can function as a simple point-to-point VPN, and with earlier versions of OpenVPN, this was the only mode. Using OpenVPN in this mode is very easy and can be setup in a matter of minutes. The down side, though, is that its uses are limited. The second mode of operation is as a server with multiple clients. This was incorporated into OpenVPN with version 2.0, and in so doing opened up a world of new possibilities for OpenVPN.

So when do we want to use OpenVPN over the previous tools we've discussed? The primary reason would be if we needed to secure all communication between client and server, regardless of protocol. This is one way in which OpenVPN distinguishes itself from OpenSSH and Stunnel. With both OpenSSH and Stunnel, we have to set up a separate tunnel for each service we want to secure. As long as we have a small number of services we want to secure, those tools do the job perfectly well. But what if we have 25 services we want to secure? Or what if we want to also secure ICMP traffic? In both of these situations, neither OpenSSH nor Stunnel meet our needs. OpenVPN, however, meets our needs quite well.

OpenVPN creates a virtual ethernet adapter which is assigned a VPN IP address. Once the connection with the server is established, any communication addressed to the VPN address of the server is encrypted.

Point-to-Point VPN

Let's say our server has an IP address of 192.168.1.1 and the client has an IP address of 192.168.1.100. The first thing we need to do is generate a shared key:

```
openvpn --genkey --secret keyfile.txt
```

Once our key is generated and we've copied it securely to both ends of our VPN, we initiate our OpenVPN session like so:

```
openvpn --config /path/to/my_config.txt
```

Example 6. Point-to-Point Server

```
remote 192.168.1.100
dev tap
ifconfig 10.1.1.1 255.255.255.0
secret /path/to/keyfile.txt
```

Example 7. Point-to-Point Client

```
remote 192.168.1.1
dev tap
ifconfig 10.1.1.2 255.255.255.0
secret /path/to/keyfile.txt
```

The config files are fairly simple. We specify the real IP address of the host on the other end of our VPN. Then we specify "dev tap." You can also use "dev tun," but in my experience, TAP works better than TUN on Windows boxes, so if you're on a heterogeneous network, I would recommend using TAP. After that, we specify our VPN address and the subnet mask. Finally, we specify the path to our shared key file. Once our VPN session is established, as long as our server talks to the client on its VPN address of 10.1.1.2, any communication will be encrypted. The same can be said for the client as long as it talks to the server on its VPN address of 10.1.1.1.

Multi-Client VPN

The multi-client VPN configuration is potentially much more complex. However, it is also substantially more versatile. We create our shared key the same way we did previously. In addition, we need to create Diffie-Hellman parameters to use during the key exchange process:

`openssl dhparam -out dh1024.pem 1024`

Finally, we need to create a certificate and key for the server and the client. As mentioned in the Stunnel section, I recommend using a tool like XCA to handle this for you.

Example 8. Multi-Client VPN - Server

```
port 1194
proto tcp-server
dev tap
ca /etc/openvpn/my_ca.crt
cert /etc/openvpn/server.crt
key /etc/openvpn/server.pem
dh /etc/openvpn/dh1024.pem
tls-auth /etc/openvpn/keyfile.txt
server 10.200.100.0 255.255.255.0
cipher AES-256-CBC
comp-lzo
verb 3
keepalive 5 120
client-to-client
```

In the above config for our server, we first specify that we are using port 1194. This is the IANA assigned port for OpenVPN. We when specify that we are the server and that we are using TCP (as opposed to UDP, which is also a valid option). Then we specify the paths to our various key and certificate files. Next we specify the address range we're going to use for our VPN. In this case, we're using 10.200.100.0; the server will take 10.200.100.1 for itself and will hand out subsequent addresses to the clients. Next, we specify the cipher we're going to use. This is an optional step, but if it is specified on the server, it must also be specified on the clients. Then we enable LZO compression and set a verbosity level of "3." This can be any number between 0 and 9. An adequate level of verbosity is generally 3 or 4. The next line says to send a ping every 5 seconds and to consider the remote peer to be down if no ping is received for 120 seconds. The last line, "client-to-client," makes it so that the VPN clients can see each other. If this is a behavior you do not want, simply leave this line out.

Example 9. Multi-Client VPN - Client

```
client
dev tap
proto tcp
remote my.openvpn.server 1194
ca /etc/openvpn/my_ca.crt
cert /etc/openvpn/client.crt
key /etc/openvpn/client.pem
tls-auth /etc/openvpn/keyfile.txt
cipher AES-256
comp-lzo
verb 3
```

The client config is much simpler than the server config. First we state that we are a client, that we are using "dev tap" and that we are using the TCP protocol. Next we specify the address and port of our OpenVPN server. After this we specify the paths to our various key and certificate files. On the following line, since we specified the cipher on the server, we have to do the same on the client. Then we enable compression and set the verbosity level to 3.

OpenVPN Conclusions

- Pros
  - OpenVPN secures all communication between server and client.
  - Compression and persistent tunnel in addition to keepalive packets partially mask the transaction length.
  - Multiple levels of authentication can be used.
  - OpenVPN uses LZO compression, which is extremely fast.
  - Very detailed logging
  - Extremely versatile
  - As long as we're using a non-privileged port, OpenVPN can be run by a non-privileged user.
- Cons
  - OpenVPN can be substantially more complex to setup than either Stunnel or OpenSSH.
  - A modest understanding of cryptography is required.

    - A modes understanding of the setup and use of Certificate Authorities is required.

There are a few points about OpenVPN that one must keep in mind. First, much more software is required on both client and server than with either Stunnel or OpenSSH. Second, for Windows clients, there is a nice GUI available that runs in the system tray. Finally, there is the issue that OpenVPN allows split tunneling. Split tunneling is when a VPN client can talk to VPN hosts and non-VPN hosts at the same time. There are dangers associated with split tunneling and warrant risk/benefit analysis before implementation.

Interesting Uses for OpenVPN

There are a number of potentially interesting uses for OpenVPN. Some of them will require some additional modifications to other things, say your localhost firewall for example. Here are a few of the interesting uses that I've been experimenting with.

- Create an administration VPN. Any work or changes that must be done as root (or as Administrator on a Windows box) goes over the VPN. Neat and tidy and substantially decreases the chances of a super user password getting out into the wild.

- Similar to the previous suggestion, create a security VPN. For example, I have all of my Snort sensors on my security VPN. This also affords me the ability to use some security tools a little more aggressively over the VPN than I would normally do over the enterprise network.

- Use OpenVPN as a WEP replacement. Take an old machine, install Linux on it, slap in a wireless card, and install wireless access point software. Then, with some creative firewall rules, you can require the client to be on the VPN before granting them full accesss to your wireless LAN.

Conclusion

I've just barely scratched the surface of the things that you can do with these tools. As with any software tools, it is a good idea to do your homework. Read the documentation, do some experimentation, and take a look at what other people have done with those tools. All software tools, especially security tools, require regular care and feeding. Be vigilant in monitoring vulnerabilities and patch where appropriate. Security tools aren't going to do you much good if they have unpatched vulnerabilities. Even though OpenVPN is a powerful tool and is fun to play with, you should probably avoid using a VPN where a simple tunnel or two will suffice. Conversely, you should avoid using multiple tunnels in a situation that would be better solved with a VPN. Take the time to choose the right tool for the right job. This can be tedious and can occasionally be quite time consuming. Careful planning, regular monitoring, and ongoing upkeep are the key ingredients in securing your network communication.

Resources

- Stunnel

- OpenSSH

- OpenVPN
- OpenVPN GUI
- OpenSSL
- XCA

About the Author

Kurt Kincaid is a Network Security Adminstrator and UNIX Administrator in Springfield, Illinois where he lives with his wife and daughter. Vice-President of his local InfraGard chapter and a self-professed InfoSec geek, he teaches Taiji Quan and Pencak Silat in his spare time.

Definitions

Term	**Definition**
Black Hat	A hacker who is chaotic, anarchistic, and breaks the law. These are the unethical hackers, commonly referred to as crackers.
IANA	Internet Assigned Numbers Authority. The authority responsible for assigning numbers, such as port or socket numbers.
IP Address	An identifier for a computer or device on a TCP/IP network. Networks using the TCP/IP protocol route messages based on the IP address of the destination. The format of an IP address is a 32-bit numeric address written as four numbers separated by periods. Each number can be zero to 255. For example, 1.160.10.240 could be an IP address.
MAC Address	Short for Media Access Control address, a hardware address that uniquely identifies each node of a network.
SSL	Short for Secure Sockets Layer, a protocol developed by Netscape for transmitting private documents via the Internet. SSL works by using a private key to encrypt data that's transferred over the SSL connection. Most web browsers (Mozilla, Internet Explorer, Netscape, Opera, Firefox, etc.) support SSL, and many Web sites use the protocol to obtain confidential user information, such as credit card numbers. By convention, URLs that require an SSL connection start with ***https:*** instead of ***http:***.
TLS	Short for Transport Layer Security, a protocol that guarantees privacy and data integrity between client/server applications communicating over the Internet. TLS supercedes and is an extension of SSL.
VPN	Short for virtual private network, a network that is constructed by using public wires to connect nodes. For example, there are a number of systems that enable you to create networks using the Internet as the medium for transporting data. These systems use encryption and other security mechanisms to ensure that only authorized users can access the network and that the data cannot be intercepted.
White Hat	A hacker who does not break the law and acts in an ethical manner.