**Explore Cryptographic Technologies**

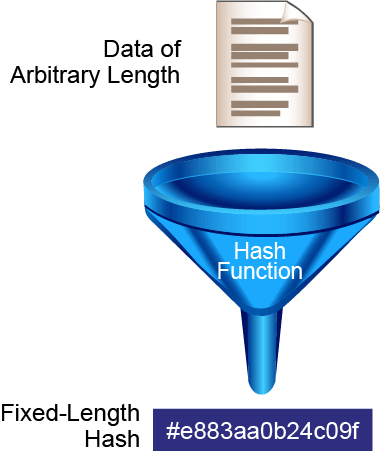
Cryptographic technologies play a very big role in today's networking. The network security analyst should be quite comfortable with the ways in which cryptographic technologies are used in modern networking.

Everyone has seen a security warning pop-up in their browser. Many people don't understand what it means when a web site or a certificate should not be trusted. In order to really understand what is going, you need a solid understanding of the underlying cryptographic technologies. Working from the top down, it requires an understanding of public-key infrastructure (PKI). Understanding PKI requires an understanding of digital signatures. Understanding digital signatures requires an understanding of asymmetric encryption and hashing.

This lab exercise will walk you through these technologies working from the bottom up. You'll start with hashing, then move on to encryption, both symmetric and asymmetric. From there, you will work with digital signatures, and finally, PKI. Practical examples will be given along the way, including the negotiation of SSH and SSL connections.

**Demonstrate Hash Algorithms**

A hash algorithm transforms data. It takes variable-length data as input and produces fixed-length data as output. Hash algorithms are designed to deterministically produce the same output when the same input is repeated, but to produce very different output when small changes are made to the input. Hash algorithms are unidirectional: that is, given the output of the hash algorithm, it is impossible to predict what the original input was. Conceptually, the computed hash value can be thought of as a “fingerprint” of the original data. The term “digest” is often used to refer to the output of a hash algorithm.



In this section of the lab exercise, you will use the Linux md5sum command to demonstrate an MD5 hashing. You will locate a 64K text file and make a copy of the file. You will verify that the two identical files have identical MD5 digests. You will then change a single character in one of the two files and recompute its MD5 digest. You will see that the modification of a single byte out of 64,000 bytes of data leads to a completely different hash fingerprint.

Task 1

Access Kali. Open a terminal window, change the directory to **/var/www/files,** and use thels –lcommand to list the directory’s contents in a long format.

**Answer**

| root@outside-srv:~#cd /var/www/files  root@outside-srv:/var/www/files#ls -l  total 44636  -rw-r--r-- 1 root root 184401 May 23 2013 55nn-X\_at\_a\_glance.pdf  -rw-r--r-- 1 root root 64988 May 23 2013 artofwar.txt  -rw-r--r-- 1 root root 13467059 May 23 2013 byod-user-experience.wmv  -rw-r--r-- 1 root root 918528 May 23 2013 calc.exe  -rw-r--r-- 1 root root 744 May 23 2013 index.php  -rw-r--r-- 1 root root 6851182 May 23 2013 ise-BYOD.flv  -rw-r--r-- 1 root root 24204476 May 23 2013 ise-videodatasheet.mp4 |
| --- |

Note the following:

* The artofwar.txt file is approximately 64K in size.

Task 2

Use the cpcommand to make a copy of the artofwar.txt file. Name the duplicate file **artofwar2.txt**. Optionally, issue the ls -l command to verify that the file artofwar2.txt was successfully created.

**Answer**

| root@sp-srv:/var/www/files#cp artofwar.txt artofwar2.txt  root@sp-srv:/var/www/files#ls -l  total 44700  -rw-r--r-- 1 root root 184401 May 23 2013 55nn-X\_at\_a\_glance.pdf  -rw-r--r-- 1 root root 64988 Sep 7 22:05 artofwar2.txt  -rw-r--r-- 1 root root 64988 May 23 2013 artofwar.txt  -rw-r--r-- 1 root root 13467059 May 23 2013 byod-user-experience.wmv  -rw-r--r-- 1 root root 918528 May 23 2013 calc.exe  -rw-r--r-- 1 root root 744 May 23 2013 index.php  -rw-r--r-- 1 root root 6851182 May 23 2013 ise-BYOD.flv  -rw-r--r-- 1 root root 24204476 May 23 2013 ise-videodatasheet.mp4 |
| --- |

Task 3

Use the md5sum command to compute the MD5 digest of the two files. You should see that the computed hash is the same, because the two files are identical.

**Answer**

| root@sp-srv:/var/www/files#md5sum artofwar.txt  0f270eabc70ed8d13c85ad6d7d4e846c artofwar.txt  root@sp-srv:/var/www/files#md5sum artofwar2.txt  0f270eabc70ed8d13c85ad6d7d4e846c artofwar2.txt |
| --- |

Note the following:

* The two files have identical contents, so they produce identical hash values.

Task 4

Make a slight modification to the file **artofwar2.txt**.

1. Use the leafpad artofwar2.txt command to open the file in the Leafpad text editor.
2. On approximately the 10th line of the file, change the capitalization of the letter **B** from “**B**y Sun Tzu” to “**b**y Sun Tzu”.
3. Save the changes with **File > Save**, and then close Leafpad with **File > Quit**.

Task 5

Compute the MD5 digest of the modified artofwar2.txt file.

**Answer**

| root@sp-srv:/var/www/files#md5sum artofwar2.txt  7343070330124014416b52efd7e2e382 artofwar2.txt |
| --- |

Note the following:

* You modified just a single character and the MD5 hash that is computed is completely different than the hash for the original file. Small changes in hash input lead to large changes in the digest. Knowing the digest does not provide any information about what the original input was.
* If you made any changes other than what was described in the directions (adding a space or a carriage return, for example), then the hash that is computed in your live lab environment will not match that shown in the example.

Task 6

MD5 produces a 128-bit digest, which is typically expressed in text format as a 32-digit hexadecimal number. MD5 works well for this demonstration, but it is not commonly used for cryptographic protection. It has been superseded by newer hash algorithms which produce larger digests. Optionally you can repeat the above steps, replacing the md5sum command with sha1sum, sha256sum, sha384sum, or sha512sum. SHA1 produces a 160-bit digest, while the number of bits produced by the other SHA family algorithms is reflected in their name.

**Hash Collisions**

Understand that it is possible that two different inputs can produce an identical hash output. Given that a hash algorithm accepts arbitrary length input and produces a relatively small, fixed-length output, it is a fact that there are more possible inputs than there are possible outputs. When two inputs produce the same hash output, it is referred to as a collision. The likelihood of a collision is dependent upon the length of the hash output. A longer output provides more potential outputs and less potential for collisions. MD5 is a relatively weak hash algorithm, with 128 bits of output. Every additional bit in the output will double the number of potential outputs. SHA-1 produces a 160-bit output. The number of possible outputs for SHA-1 is 2^32 (more than 4 billion) times greater than MD5. Hence a collision is astronomically less likely with SHA-1 than it is with MD5. SHA-256, SHA-384, and SHA-512 each provides a further decrease in the potential for collisions.

MD5 not only produces a relatively small digest: it also has known vulnerabilities. Under the scrutiny of the cryptographic research community, methods of producing files with colliding MD5 digests have been developed and published. In 2005, Xiaoyun Wang and Hongbo Yu of Shandong University in China published a paper which described an algorithm to find two different sequences of 128 bytes with the same MD5 hash. Building on Wang and Yu’s work, and the work of other cryptographic researchers, Peter Selinger of Dalhousie University published a C programming library and procedure for producing two executable programs that have the same MD5 digest, but different program behaviors.

**Demonstrate Symmetric Encryption**

Modern encryption uses secret keys and public algorithms. Using a secret algorithm will eventually lead to failure. Once the algorithm is reverse-engineered, then there is not an easy fix. With modern encryption, if a key is compromised, the key can be modified without modifying anything else in the system. Regarding network communications, the goal is to dynamically change keys at regular intervals, mitigating real-time decryption of data and minimizing the amount of data that is protected with any single key.

Encryption algorithms that use the same key for both encryption and decryption are called symmetric encryption algorithms. They are also referred to as shared key algorithms. For two systems to communicate with symmetric encryption, they must both possess (share) the common encryption key. This section of the lab will provide a simple demonstration of symmetric encryption and decryption using 128-bit AES as the cipher.

Task 1

Return to Kali. Using a terminal window, change the directory to **/var/www/files**, and display the contents with the ls –l command.

**Answer**

| root@sp-srv:~#cd /var/www/files  root@sp-srv:/var/www/files#ls -l  total 44700  -rw-r--r-- 1 root root 184401 May 23 2013 55nn-X\_at\_a\_glance.pdf  -rw-r--r-- 1 root root 64988 Feb 5 20:56 artofwar2.txt  -rw-r--r-- 1 root root 64988 May 23 2013 artofwar.txt  -rw-r--r-- 1 root root 13467059 May 23 2013 byod-user-experience.wmv  -rw-r--r-- 1 root root 918528 May 23 2013 calc.exe  -rw-r--r-- 1 root root 744 May 23 2013 index.php  -rw-r--r-- 1 root root 6851182 May 23 2013 ise-BYOD.flv  -rw-r--r-- 1 root root 24204476 May 23 2013 ise-videodatasheet.mp4 |
| --- |

Notice that the two artofwar files that you worked with previously are still here.

Task 2

Use the diff command to compare the **artofwar.txt** and the **artofwar2.txt** files.

**Answer**

| root@sp-srv:/var/www/files#diff artofwar.txt artofwar2.txt  11c11  < Hello world !  ---  > hello world ! |
| --- |

As you should remember from earlier in the lab exercise, there is a difference at line number 1. One uses a capital **H** and the other uses a lowercase **h** in the word "Hello."

Task 3

Use the openssl command to encrypt the artofwar.txt file using 128-bit AES cipher block chaining, and store the encrypted file as **artofwar.crypt**. Specify **Cisco123!** as the encryption password. You will have to enter the password a second time for verification.

**Answer**

| root@sp-srv:/var/www/files#openssl enc -aes-128-cbc -in artofwar.txt -out artofwar.crypt  enter aes-128-cbc encryption password: Cisco123!  Verifying - enter aes-128-cbc encryption password: Cisco123! |
| --- |

Note that this demonstration is not the optimal use of 128-bit AES. Optimally, a highly randomized 128-bit value is used as the key. Cisco123! is neither highly randomized nor 128 bits. OpenSSL uses an algorithm to convert the password into the appropriate length that is required for the key material.

Task 4

Display the list of files in the directory again.

**Answer**

| root@sp-srv:/var/www/files#ls -l  total 44764  -rw-r--r-- 1 root root 184401 May 23 2013 55nn-X\_at\_a\_glance.pdf  -rw-r--r-- 1 root root 64988 Feb 5 20:56 artofwar2.txt  -rw-r--r-- 1 root root 65008 Feb 5 21:03 artofwar.crypt  -rw-r--r-- 1 root root 64988 May 23 2013 artofwar.txt  -rw-r--r-- 1 root root 13467059 May 23 2013 byod-user-experience.wmv  -rw-r--r-- 1 root root 918528 May 23 2013 calc.exe  -rw-r--r-- 1 root root 744 May 23 2013 index.php  -rw-r--r-- 1 root root 6851182 May 23 2013 ise-BYOD.flv  -rw-r--r-- 1 root root 24204476 May 23 2013 ise-videodatasheet.mp4 |
| --- |

Note the following:

* OpenSSL created a new file named artofwar.crypt.
* Optionally, you can use the cat artofwar.crypt command to display the contents of the file. It is no longer a text file. The output of the cat command will not be readable.

Task 5

Use the openssl command to decrypt the artofwar.crypt file using 128-bit AES cipher block chaining, and store the decrypted file as **artofwar3.txt**. Specify **Cisco123!** as the encryption password.

**Answer**

| root@sp-srv:/var/www/files#openssl enc -d -aes-128-cbc -in artofwar.crypt -out artofwar3.txt  enter aes-128-cbc decryption password:Cisco123! |
| --- |

Task 6

List the files in the directory one last time.

**Answer**

| root@sp-srv:/var/www/files#ls -l  total 44828  -rw-r--r-- 1 root root 184401 May 23 2013 55nn-X\_at\_a\_glance.pdf  -rw-r--r-- 1 root root 64988 Feb 5 20:56 artofwar2.txt  -rw-r--r-- 1 root root 64988 Feb 5 21:14 artofwar3.txt  -rw-r--r-- 1 root root 65008 Feb 5 21:03 artofwar.crypt  -rw-r--r-- 1 root root 64988 May 23 2013 artofwar.txt  -rw-r--r-- 1 root root 13467059 May 23 2013 byod-user-experience.wmv  -rw-r--r-- 1 root root 918528 May 23 2013 calc.exe  -rw-r--r-- 1 root root 744 May 23 2013 index.php  -rw-r--r-- 1 root root 6851182 May 23 2013 ise-BYOD.flv  -rw-r--r-- 1 root root 24204476 May 23 2013 ise-videodatasheet.mp4 |
| --- |

Note the following:

* The file artofwar3.txt was created by the OpenSSL decryption activity.

Task 7

If this process worked, the artofwar.txt and artofwar3.txt files should be identical. Use the diff command to verify that their contents are identical.

**Answer**

| root@sp-srv:/var/www/files#diff artofwar.txt artofwar3.txt |
| --- |

Note the following:

* If the diff command produces no output; it indicates that there are no differences between the content of the two files.

Task 8

Optionally, you can compute the hash values of the two files using md5sum or any of the sha\*sum commands. The hash values should, of course, be identical.

Task 9

Optionally, you can repeat this process using different ciphers. The openssl list-cipher-algorithms command will display the ciphers that are available.

**Demonstrate Asymmetric Encryption**

Asymmetric encryption differs from symmetric in that two keys are used. The two keys are uniquely paired to each other. Anything that is encrypted with one of the paired keys requires the other paired key to decrypt. Normally a system will generate a key pair, and it will keep one of the keys totally secret (the private key) and it will make the other key freely available (the public key). The term public key encryption is often used interchangeably with asymmetric encryption.

Public key encryption can be used in different ways. Imagine a system—call it SystemA. SystemA generates a public/private key pair and publishes its public key. If any other system wants to send a private message to SystemA, that other system can encrypt the message using SystemA’s public key. Since only SystemA has the corresponding private key, only SystemA can read the message.

On the other hand, imagine that SystemA wants to send a message to another system and wants the other system to be certain that the message that is originated from SystemA. SystemA can then encrypt the message with its private key. Any other system can decrypt the message using SystemA’s public key, which does not provide privacy. But the fact that the message can be decrypted with SystemA’s public key proves that the message was encrypted with SystemA’s private key. Since only SystemA has that private key, the message must have originated from SystemA. This method provides origin authentication and non-repudiation.

Another point to understand about symmetric versus asymmetric encryption is that asymmetric encryption is much more computationally expensive than symmetric encryption. Therefore, virtually all secure communication protocols rely on symmetric encryption for bulk data transfer.

A challenge with symmetric encryption is getting both sides to know the shared key in a secure fashion. SSH version 1 tackles this challenge in a clever way. When an SSH client connects to an SSH server, they perform some negotiation in clear text and then the server presents its public key. The client then generates a shared session key to be used for symmetric encryption of this connection. The client encrypts the shared session key with the server’s public key. The client then sends the encrypted session key to the server. Only the server has the corresponding private key to decrypt the session key. The server decrypts the shared session key, and symmetric encryption is then used. It is important to note that the public key is also used to recognize the server is really the server and not a man in the middle. SSH clients maintain a known hosts repository which contains the public keys of authorized SSH servers. A challenge with SSH, though, is the authorization process. Generally, when an SSH client connects to a server for the first time, it will present the key to the user. If the user accepts the key, it is then stored in the known hosts repository. Assuming the first visit was to the authentic server, this process works well. But if the first visit to a server is intercepted by a man in the middle, the user may accept the wrong key.

In this section of the lab exercise, you will enable debugging of SSH connections on the router. Then you will establish an SSH connection from the Kali. You will then examine the debug output to see the key negotiation process.

Task 1

Establish Telnet session to the router

**Answer**

| root@sp-srv:~#telnet 209.165.200.225  Trying 209.165.200.225...  Connected to 209.165.200.225.  Escape character is '^]'.  User Access Verification  Username: admin  Password: Cisco123!  Internet-Rtr# |
| --- |

Task 2

Enable debugging of IP SSH on the router with the debug ip ssh command. Also, use the terminal monitor command to display logging messages to this Telnet session.

**Answer**

| Internet-Rtr#debug ip ssh  Incoming SSH debugging is on  Internet-Rtr#terminal monitor |
| --- |

Task 3

From a new terminal window, initiate an SSH connection to the router. Specify **SSH version 1**. SSH version 2 uses a different negotiation method and produces less informative debug output on IOS SSH servers.

**Answer**

Note the following:

* In the example syntax below, the first argument is **-1** (a minus sign followed by the numeral 1), which specifies that SSH version 1 will be used.
* In the example syntax below, the second argument is **-l admin** (a minus sign followed by a lowercase L, and admin), specifying the local user ID.
* Since the SSH client on the Kali has never seen the public key of the router, it displays the key’s fingerprint and requires you to verify that it is the correct key. Once you verify that the key belongs to router, the SSH client will store this information in its known hosts repository.

| root@sp-srv:/var/www/files#ssh -1 -l admin 209.165.200.225  The authenticity of host '209.165.200.225 (209.165.200.225)' can't be established.  RSA1 key fingerprint is 38:b0:64:e4:29:55:37:5e:ac:87:52:03:6b:9f:d0:d0.  Are you sure you want to continue connecting (yes/no)? yes  Warning: Permanently added '209.165.200.225' (RSA1) to the list of known hosts.  admin@209.165.200.225's password: Cisco123!  This is the router for the pod.This is the router for the pod.  Internet-Rtr# |
| --- |

Task 3

Use the exit command to terminate the SSH connection.

**Answer**

| Internet-Rtr#exit  Connection to 209.165.200.225 closed. |
| --- |

Task 4

Return to the Telnet connection to the router. You should find debug output that is associated with the SSH connection. Take a moment to review the debug output and verify that the SSH version 1 connection process proceeded as described in the introduction to this section of the lab. Sample debug output with annotations is provided below.

**Answer**

| 00:25:35: SSH1: starting SSH control process |
| --- |
| The Rtr receives the SSH connection request from the SP-Srv. |

| 00:25:35: SSH1: sent protocol version id SSH-1.99-Cisco-1.25 |
| --- |
| The Rtr announces that it supports SSH version 1.99 (SSH 2.0 with the ability to fall back to version 1.5). |

| 00:25:35: SSH1: protocol version id is - SSH-1.5-OpenSSH\_6.0p1 Debian-4+deb7u2 |
| --- |
| Kali identifies its SSH version, it can support version 1.5. |

| 00:25:35: SSH1: SSH\_SMSG\_PUBLIC\_KEY msg  00:25:44: SSH1: SSH\_CMSG\_SESSION\_KEY msg - length 144, type 0x03 |
| --- |
| The Rtr passes its public key to the SP-Srv. |

| 00:25:44: SSH1: RSA decrypt started  00:25:44: SSH1: RSA decrypt finished  00:25:44: SSH1: RSA decrypt started  00:25:44: SSH1: RSA decrypt finished |
| --- |
| The Kali provides session key data encrypted with the Rtr’s public key. The Rtr decrypts that data using its private key. |

| 00:25:44: SSH1: sending encryption confirmation  00:25:44: SSH1: keys exchanged and encryption on |
| --- |
| The Rtr is using the dynamic session key that is provided by the Kali. At this point, all data that is transmitted between the two systems is protected. |

| 00:25:44: SSH1: Installing crc compensation attack detector.  00:25:44: SSH1: SSH\_CMSG\_USER message received  00:25:44: SSH1: authentication request for userid admin  00:25:44: SSH1: SSH\_SMSG\_FAILURE message sent  00:25:49: SSH1: SSH\_CMSG\_AUTH\_PASSWORD message received  00:25:49: SSH1: authentication successful for admin |
| --- |
| The authentication process was completed after the connection was secured. |

| 00:25:49: SSH1: requesting TTY  00:25:49: SSH1: setting TTY - requested: length 24, width 80; set: length 24, width 80  00:25:49: SSH1: SSH\_CMSG\_EXEC\_SHELL message received  00:25:49: SSH1: starting shell for vty |
| --- |
| An EXEC shell is provided for the SSH connection. |

| 00:25:57: SSH1: Session terminated normally |
| --- |
| The SSH session terminated normally. |

Task 5

Turn off debugging of IP SSH on the Internet-Rtr.

**Answer**

| Internet-Rtr#undebug all  All possible debugging has been turned off |
| --- |

**Create a Key Pair and Digital Signature**

At this point, you should understand that asymmetric encryption can be used for origin authentication. If SystemA encrypts a message with its private key, any other system that can decrypt the message with SystemA's public key knows that SystemA did the encryption. You should also understand that hashing technology provides fingerprints of data. Digital signatures leverage these two concepts. SystemA can sign a document by first hashing the document to produce its fingerprint and then encrypting the hash digest with its private key. Other systems can verify whether the document has changed since SystemA signed to document. To verify a signature, another system will compute the hash value of the document they received and they will decrypt the signature with SystemA's public key. If the computed hash matches the decrypted signature, then the document hasn't changed. The signature is valid. If the two values do not match, then something in the document has changed. The signature is invalid.

In this section of the lab, you will create an RSA public/private key pair on the Kali, and create a signature for the artofwar.txt file.

Task 1

On the Kali, return to the terminal window where you are in the **/var/www/files** directory.

Task 32

Use the openssl command to generate a 4096-bit RSA private key, protecting the key with 128-bit AES and the pass phrase Cisco123! Store the key in a file named **private.pem**.

**Answer**

| root@sp-srv:/var/www/files#openssl genrsa -aes128 -passout pass:Cisco123! -out private.pem 4096  Generating RSA private key, 4096 bit long modulus  ....................................................................................................................++  .............................................................................................................++  e is 65537 (0x10001) |
| --- |

Note the following:

* The private key must remain private. If any other system obtains the private key, it can be used to impersonate the legitimate owner of the private key.
* Encrypting the private key with AES and a pass phrase provides an extra layer of protection for the key.
* Any use of the private key will require the specification of the pass phrase. In this case, the pass phrase is Cisco123!

Task 2

Use the openssl command to create the public key that is paired with the private key that is stored in the **private.pem** file. Store the public key in the file named **public.pem**.

**Answer**

| root@sp-srv:/var/www/files#openssl rsa -in private.pem -passin pass:Cisco123! -pubout -out public.pem  writing RSA key |
| --- |

Note the following:

* To access private key that is stored in private.pem, the pass phrase **Cisco123!** was required.
* In normal usage of public/private key pairs, there is no need to encrypt the public key. Systems will commonly publish their public keys for everyone to see.

Task 3

Use the ls command to verify that the two key files have been created.

**Answer**

| root@sp-srv:/var/www/files#ls  55nn-X\_at\_a\_glance.pdf artofwar.txt ise-BYOD.flv  artofwar2.txt byod-user-experience.wmv ise-videodatasheet.mp4  artofwar3.txt calc.exe private.pem  artofwar.crypt index.php public.pem |
| --- |

Task 4

Use the sha256sum command to generate a 256-bit digest of the artofwar.txt file, and redirect the output to a file named **artofwar.digest**. Use the ls command to verify that the file is created.

**Answer**

| root@sp-srv:/var/www/files#sha256sum artofwar.txt > artofwar.digest  root@sp-srv:/var/www/files#ls  55nn-X\_at\_a\_glance.pdf artofwar.txt ise-videodatasheet.mp4  artofwar2.txt byod-user-experience.wmv private.pem  artofwar3.txt calc.exe public.pem  artofwar.crypt index.php  artofwar.digest ise-BYOD.flv |
| --- |

Task 5

Use the openssl command rsa utility to encrypt (sign) the artofwar.digest file using the private key, and store the output in the file **artofwar.sig**. Use the ls command to verify that the file is created.

**Answer**

| root@sp-srv:/var/www/files#openssl rsautl -sign -inkey private.pem -in artofwar.digest -out artofwar.sig  Enter pass phrase for private.pem: Cisco123!  root@sp-srv:/var/www/files#ls  55nn-X\_at\_a\_glance.pdf artofwar.sig ise-BYOD.flv  artofwar2.txt artofwar.txt ise-videodatasheet.mp4  artofwar3.txt byod-user-experience.wmv private.pem  artofwar.crypt calc.exe public.pem  artofwar.digest index.php |
| --- |

Task 6

Use the openssl rsa utility to decrypt (verify) the artofwar.sig file. Then use the sha256sum command to produce the SHA-256 digest of the **artofwar.txt** file. The two outputs match, so the signature is valid.

**Answer**

| root@sp-srv:/var/www/files#openssl rsautl -verify -inkey public.pem -pubin -in artofwar.sig  aed8359f1c5012c0f3ead7449815dbba2475bcd385ae84db6e360edde46929db artofwar.txt  root@sp-srv:/var/www/files#sha256sum artofwar.txt  aed8359f1c5012c0f3ead7449815dbba2475bcd385ae84db6e360edde46929db artofwar.txt |
| --- |

Task 7

Remember that you changed one character from upper case to lower case in the artofwar2.txt file. Use the sha256sum command to produce the SHA-256 digest of the artofwar2.txt file. Note that this digest does not match the decrypted signature. Any tampering with the file contents will void the signature.

**Answer**

| root@sp-srv:/var/www/files#sha256sum artofwar2.txt  bcef5905e5b5836ae03050eb57ca57f2db77e22993fd6a3cf21a53b0b2d70b3a artofwar2.txt |
| --- |

**Explore Public-Key Infrastructure**

While SSH makes clever use of asymmetric encryption to protect the shared session key used for symmetric encryption, it puts the burden of verifying the server’s public key on the user. SSL, commonly used to protect HTTPS sessions, uses asymmetric encryption in a similar way as SSH version 1, but it uses public key infrastructure to remove the burden of verifying the server’s public key from the user. PKI provides a system to authenticate the web server's public key. With PKI, the web server enrolls with a certificate authority to obtain an identity certificate. The identity certificate specifies essential information about the server, including the server’s public key. The CA produces the certificate and digitally signs the certificate. To sign, the CA hashes all the data in the identity certificate and then encrypts that hash with its private key. Any system that has the CA’s public key can verify the signature on the certificate by hashing the data in the certificate and decrypting the CA signature with the CA’s public key. If the computed hash and the decrypted signature match, then the data in the certificate is what was signed by the CA. The web browser maintains a cache of root CA certificates containing the public keys of those CAs. PKI can be deployed with CAs in a hierarchy. A root CA signs the certificates of subordinate CA's. When hierarchical PKI is used, the server will provide its own certificate and a chain of certificates to the web browser. The browser will use its own copy of the root CA public key to verify the signature of the intermediate CA certificate, and then it will use the public key from the intermediate CA certificate to verify the signature of the server certificate.

Task 1

Search for and launch **Internet Explorer** from the Windows **Start** menu. Do not use Firefox or Edge for this demonstration. All three browsers behave similarly, but Internet Explorer makes it easier to view a certificate chain.

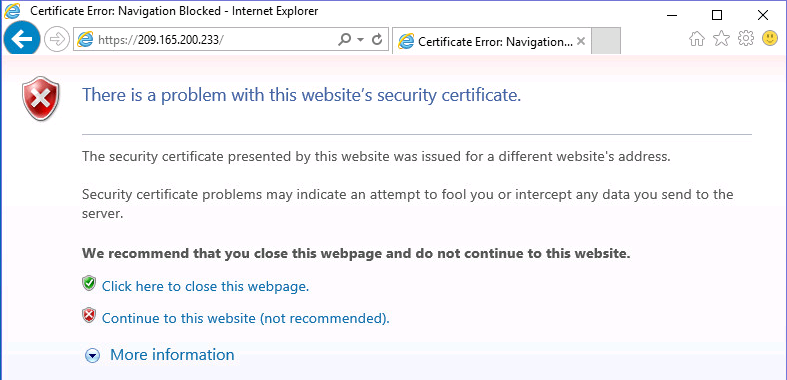
Task 2

First, browse **https://www.google.com**, which should succeed without any security warnings.

Task 3

Browse [**https://209.165.200.233**](https://209.165.200.233) **(just an example)**, which will result in a security warning. Examine the security warning.

**Answer**



Note the following:

* The message reads: The security certificate that is presented by this website was issued for a different website's address. The certificate is for http://www.services.public, but you visited 209.165.200.233.
* The three most common reasons for browsers to display security warnings are the following:
  1. The hostname in the URL entry field does not match name fields in the identity certificate. DNS is critical for PKI.
  2. The current date is outside the range that is specified by the certificate’s valid from and valid to fields. Accurate clock and calendar functions are critical for PKI.
  3. The browser could not verify the certificate’s signature, which usually happens when the browser does not have the root CA’s certificate within its certificate store, such as when the server is using a self-signed certificate. Another possibility is that the certificate was forged by an attacker who does not have access to the root CA’s private key.

Task 4

Browse **https://www.google.com**. The specified hostname will match that specified in the certificate, so the HTTPS connection establishes without security warnings.

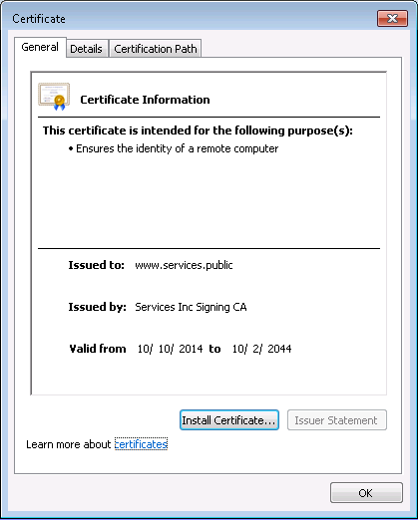
Task 5

Click the **lock** icon on the right side of the URL entry field to display the website identification information.

Task 6

In the Website Identification panel, click **View Certificates**. A certificate window opens. Examine what is presented under the **General** tab of the Certificate window.

**Answer**



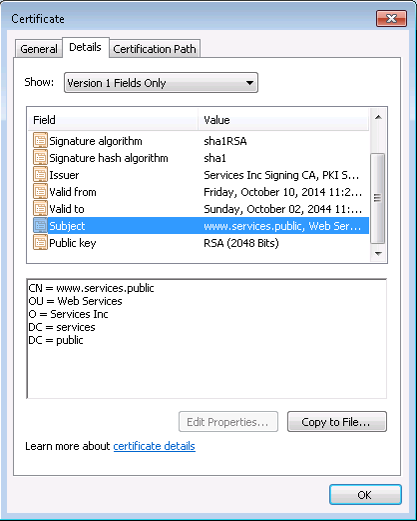
Note the following:

* The **Issued to** field specifies the entity that is identified with this certificate.
* The **Issued by** field specifies the CA which signed this certificate.
* The **Valid from** and **Valid to** fields identify the time range for which the CA server will maintain revocation information that is associated with this certificate. Browsers will display a security warning if the date is outside of this range.

Task 7

Select the **Details** tab in the Certificate window. Examine the fields that are contained within the certificate. Selecting a field will display the fields contents. Select the **Subject** field and examine the data that it contains.

**Answer**



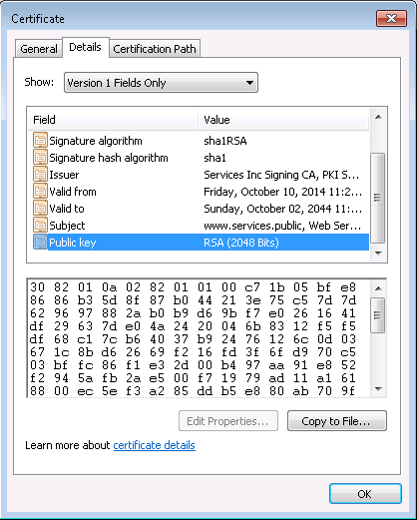
Note the following:

* **CN** specifies the common name of the identified entity. In this case, it is http://www.google.com. The browser expects this name to match the host name that is specified in the URL entry field. Some certificates will include a SAN, which can specify alternate names that will also be accepted by the browser.
* **OU** specifies the organizational unit, which is generally a department within a company, but it can be any subdivision of the organization.
* **O** specifies the organization, which is generally a company, but can be other types of organizations such as governments or associations.
* **DC** specifies domain components.

Task 8

Select the **Public key** field and examine its contents.

**Answer**



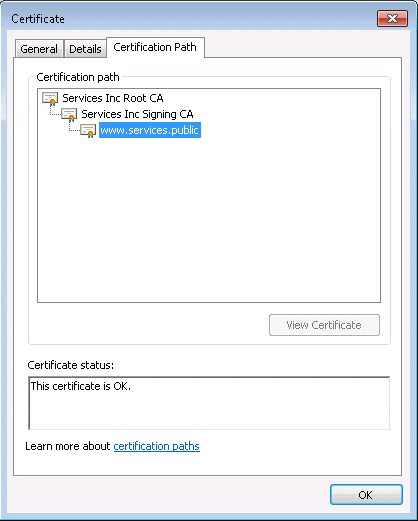
Note the following:

* This field is one of the most important fields in an identity certificate. If a client can verify the signature on the certificate, then it knows that this signature is indeed the public key that is owned by the identified server. If the client encrypts a message using this public key, it knows that only the identified server has the associated private key that can decrypt the message. In the case of SSL, the client will generate a shared session key and encrypt it with this public key. The only system that can decrypt the message is the one that was issued this identity certificate.
* Normally, the public key is visible to everyone. Attackers can copy this identity certificate to a file and possess the server’s public key. They may establish themselves as a man in the middle and present this certificate to a client. Since the signature is valid, the client will accept the certificate. But when the client encrypts a proposed shared session key with this public key and sends it to the attacker, it is useless to the attacker. The attacker does not have the real server’s private key.

Task 9

Click the **Certification Path** tab. Examine what is displayed.

**Answer**



Note the following:

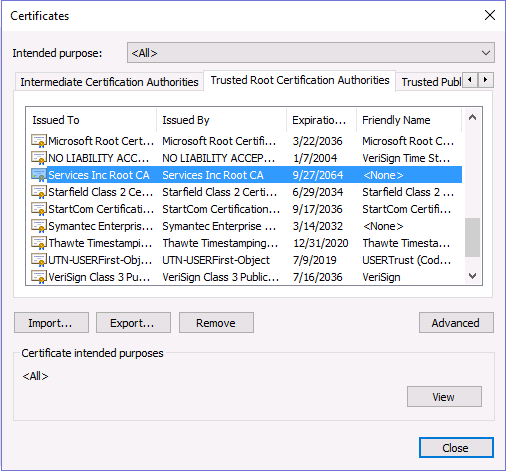
* There is a CA hierarchy. The certificate was signed by the Services Inc Signing CA. The Services Inc Signing CA’s certificate was signed by the Services Inc Root CA.
* The certificate at the top of the hierarchy will be self-signed.
* For a browser to verify the signature on a certificate, it must have the certificate of at least one of the CAs in this hierarchy. The browser’s certificate store is where the trust relationship begins.

Task 10

Close the **Certificate** window. View the trusted root CA certificates that are in Internet Explorer’s certificate store:

1. Select **Internet Options** from Internet Explorer’s **Settings** menu (use the gear-shaped icon in the upper right corner to access this menu).
2. Select the **Content** tab in the Internet Options window.
3. Click the **Certificates** button to open the Certificates window.
4. Select the **Trusted Root Certification Authorities** tab. Examine the list of trusted root certificates.

**Answer**



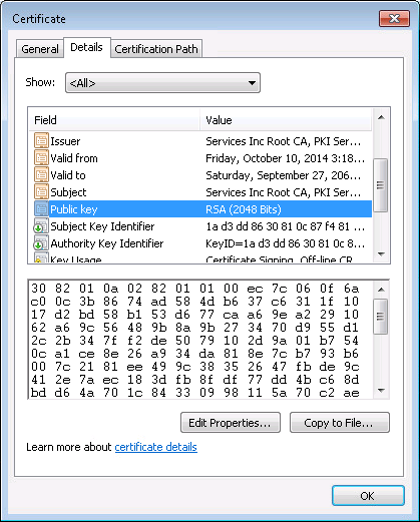
Note the following:

* Browser vendors work with public CA vendors to maintain current root certificates of valid CAs in their certificate stores. You can see certificates from well-known vendors such as VeriSign, Thawte, GoDaddy, and GeoTrust.
* Toward the bottom of the list is the Services Inc Root CA certificate. This certificate was added in the lab environment. It is common for enterprises to implement their own CA hierarchies. When they do so, it is optimal for them to manage browser installations on corporate systems, ensuring that their private CA hierarchy is trusted by the browsers on corporate systems.

Task 10

Select the **Services Inc Root CA** certificate in the list and click **View**. Select the **Details** tab on the Certificate Window.

**Answer**



Task 11

Consider how Internet Explorer validated the https://www.google.com certificate.

**Capture Packets from an SSL/TLS Connection**

Take one more look at SSL/TLS, this time by analyzing a packet capture. In this section of the lab exercise, you will quickly perform the capture. In the next section of the lab exercise, you will analyze the capture. Basically, you are going to close your browser, launch Wireshark, start a capture, launch a browser and browse https://www.google.com, and stop the capture. The more quickly you go through these steps, the less spurious traffic will be included in the capture.

Task 1

Launch **Wireshark** on Outside-Win.

Task 2

Start a capture. Select **Capture** > **Options**, then select the **Ethernet** interface and click **Start**.

Task 3

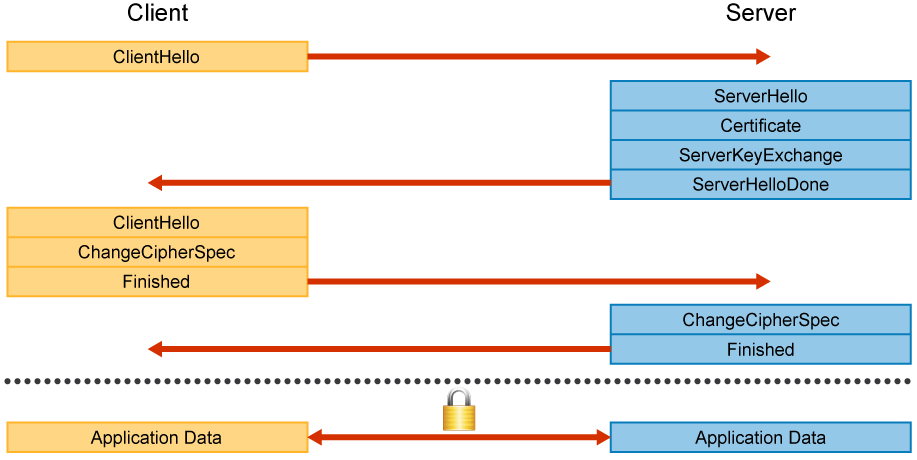
Launch **Firefox**, and browse [**https://www.google.com**](https://www.google.com).

Task 4

Return to **Wireshark** and select **Capture** > **Stop**.

**Analyze SSL/TLS Negotiation**

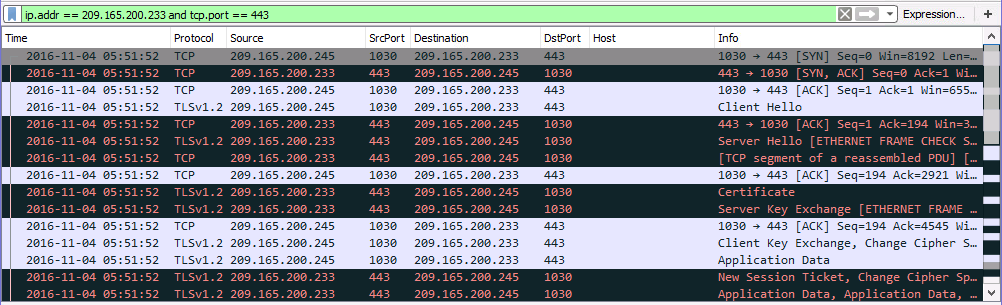
Now it's time to look at the capture. There is a lot of complexity in the SSL/TLS protocols, and in the cryptographic algorithms employed. The intent is not to make you an expert in them here. But it is good for the analyst to see how the negotiation works across the network. You will see that negotiation follows this process:



Task 5

Examine the packet summary table. There is likely spurious traffic captured along the SSL/TLS conversation. Use Wireshark’s display filter feature to remove the spurious traffic from the summary display. Enter **ip.addr == a.b.c.s and tcp.port == 443** in the display filter field immediately above the packet summary pane. When this display filter is applied, the packet summary should display only packets where one of the two IP addresses is a.b.c.d (google.com ip) and one of the two TCP ports is 443 (the standard port for SSL/TLS).

**Answer**



Note the following:

* As you would expect with any normal TCP connection, the first three packets are associated with the TCP three-way handshake.
* Note the description of the SSL negotiation packets as shown in the Info column. The following are seen: Client Hello, Server Hello, Certificate, Server Key Exchange, and Client Key Exchange.
* After that sequence, there are packets that are associated with TCP connection maintenance and the transfer of application data. The application data is protected by the TLS tunnel.

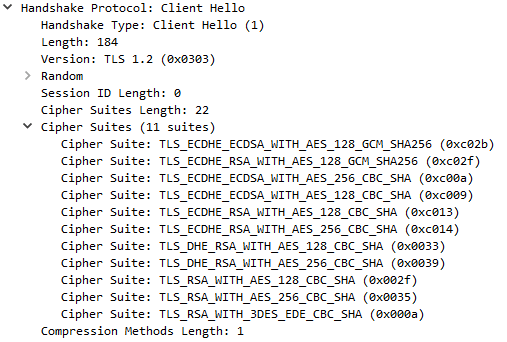
Task 6

Select the **Client Hello** packet. Expand the SSL header: **Secure Sockets Layer** > **TLSv1.2 Record Layer…** > **Handshake Protocol: Client Hello**. Note that the client is requesting a TLS version 1.2 session.

Task 7

Expand the **Cipher Suites (11 suites)** field and examine its contents.

**Answer**



Note the following:

* This list of cipher suites that the client is willing to accept. It is also in order of preference by the client.
* Across the 11 suites, four key exchange methods are offered:
  1. **TLS\_RSA:** The client chooses a random session key value and encrypts it with the server's RSA public key.
  2. **TLS\_DHE\_RSA:** Diffie-Hellman key exchange where the client encrypts the materials it exchanges using the server's RSA public key.
  3. **TLS\_ECDHE\_RSA:** Similar to TLS\_DHE\_RSA, but using the elliptical curve variant of Diffie-Hellman.
  4. **TLS\_ECDHE\_ECDSA:** Similar to TLS\_ECDHE\_RSA, but with an elliptical curve that is based DSA key pair instead of an RSA key pair.
* Across the 11 suites, four bulk encryption algorithms are offered:
  1. **3DES\_EDE\_CBC:** 168 bits of keying material that is applied by the 56-bit DES algorithm three times. In certain circumstances, which only provides 112 bits worth of effective strength.
  2. **AES\_128\_CBC:** 128-bit encryption keying material, cipher block chaining.
  3. **AES\_256\_CBC:** 256-bit encryption keying material, cipher block chaining.
  4. **AES\_128\_GCM\_SHA256:** 128-bit keying material, galois/counter mode. 256-bit integrity bits. Note, with GCM, the hash-based integrity is built into the encryption algorithm.
* Across the 11 suites, two data integrity algorithms are offered:
  1. **SHA:** 160-bit digest
  2. **SHA256:** 256-bit digest. Note, if the encryption algorithm uses GCM mode, the integrity is built into the encryption algorithm and is not a separate computation.

Task 8

Select the **Server Hello** packet. Expand the SSL header: **Secure Sockets Layer > TLSv1.2 Record Layer… > Handshake Protocol: Server Hello**. Note that the server is informing the client which cipher suite was selected: TLS\_ECDHE\_RSA\_WITH\_AES\_128\_GCM\_SHA256.

Task 9

Select the **Certificates** packet. Expand the SSL header: **Secure Sockets Layer** > **TLSv1.2 Record Layer…** > **Handshake Protocol: Certificate** > **Certificates**.

Task 10

Note that there are four certificates. Drill down to see the certificates that contain the same fields that you saw in Internet Explorer, including the subject, issuer, signature, and subject public key. Note that one of the certificates is included twice. The server offered its certificate and it offered the certificate chain from the server's certificate through the signing CA to the root CA.

Examine the server key exchange and client key exchange packets. Each system is offering the other their EC Diffie-Hellman parameters. Each will take their private ECDH parameters and the public parameters from the peer and compute the same shared keying material to use with symmetric encryption. At this point, they enable the cipher specification. All data that is associated with this connection, from this point on, is protected with 128-bit AES and 256-bit SHA.