The Slapper Worm as an example of a vulnerability caused by a buffer overflow.

GIAC Certified Incident Handler (GCIH) Practical Version 2.1
OPTION 2 – Support for the Cyber Defense Initiative
Submitted by Craig Brown

Abstract

This paper focuses on the Slapper worm and how it compromises a victim's system. The Slapper exploit is made possible by a buffer overflow during an SSL handshake between a client and an SSL enabled server

I've been a software developer for twenty years and have witnessed the evolution of many programming styles. It is my intent to contrast the traditional application programming mindset (which lead to the buffer overflow and intern the creation of Slapper) with a mindset that is required to produce vulnerability free internet applications. I use a recent Apache exploit as an example.

There are many kinds of "defects" that can be found in software applications. Traditionally, the worst of these defects would result in data loss. Before the internet, data loss was considered the worst type of defect and a great deal of "due diligence" has been incorporated into the software development process so that if a defect is encountered, data loss will be minimized.

Programming software applications for the internet requires a different mind-set then that of the traditional software design methodology. Probably the most common flaw in "C" programming is the buffer overflow. If a programmer instructs a computer to insert ten bytes of data into a five byte data buffer, the CPU will fill the buffer with the first five bytes and then over write the next five bytes (which may belong to another part of the program). At a time when computer memory was only a few thousand bytes, an overflow like this would crash the system. With the many megabyte systems of today, the results are unpredictable. In short, when writing a "stand alone" application, the worst thing that can happen is data loss or a system crash.

With internet application, the risks are much higher. A buffer overflow can result in hostile foreign code being executed on a system. The hacker's intent may be to crash the system or something much more heinous. In the age of internet programming, a developer must take precautions and expect the unexpected. Any write operation must check both it's buffer size and the size of the data being passed. When using a software toolkit, no assumption should be made that the toolkit writer included error prevention and error handling.

There is a major risk when transmitting confidential information across the web. Text that has been entered into web-based forms can be easily read as it travels across the internet towards it's destination. This risk can be mitigated by securing a web session with the SSL protocol. SSL is difficult to implement and in order to understand it, there are many concepts that must first be understood. These concepts include digital certificates, public keys, private keys, symmetric and asymmetric encryption (just to name a few).

Developers frequently use toolkits to implement SSL functionality. The most popular SSL toolkit is OpenSSL. Unfortunately, there is a known defect in some versions of OpenSSL that will allow a buffer overflow. This vulnerability has been exploited with the Linux Slapper worm. The Slapper worm allows a hacker to control a remote system and perform many different nefarious tasks. The availability of exploits (such as the Slapper worm) could have been avoided if the original programmers had not made any "assumptions" and tested all parameters that were being passed.

Introduction

This paper is being written as a requirement for the GCIH certification. I have received permission from GIAC[1] to modify the original assignment in the following way. It is my intent to not only document a vulnerability but to also attempt to explain the consequences of using a particular software development methodology in the context of avoiding future similar vulnerabilities. It is my hope that I can share my experiences with other security professionals by using my development experience to create a document that will provide some insight into "why things are the way they are" (while still meeting the core requirements of the original assignment). The majority of opinions expressed in this paper are my own.

The exploits discussed in this paper all relate to vulnerabilities that have been found in software that is using the OpenSSL library. The biggest exploit thus far has been termed "The Linux Slapper Worm". Slapper uses a vulnerability in OpenSSL to install and compile code on a victim's system. Once installed, any number of vulnerabilities and behaviors can be installed into the system. The vulnerability that makes all this possible was very easy to fix. It was the result of a "misunderstanding" between the person that created the low level code and the person that used it in an application.

Greatly simplified, the vulnerability occurred as follows. The Apache server is composed of many software modules. The module that provides SSL functionality is mod_ssl. mod_ssl is also composed of multiple software components. It's SSL functionality is derived from the OpenSSL toolkit. There is a defect in a particular routine (in OpenSSL) where an error will cause a buffer overflow. A procedure in mod_ssl uses this routine and causes this error. Essentially, The developer of the OpenSSL routine assumed that the routine caller (in this case the person that wrote mod_ssl) would test the size of a parameter that is being passed to it. The person that wrote mod_ssl either assumed that the OpenSSL programmer is checking for parameter size or did not think about it at all. The resulting buffer overflow is a combination of errors.

Buffer Overflows

The vulnerabilities discussed in this paper are exploited through buffer overflows induced by bad parameter passing. As the name implies, a buffer overflow is an error that occurs (within a running software application) when a certain amount of

[1] Reviewed by Patrick Prue [pprue@cogeco.ca]

computer memory has been allocated to hold an amount of data that is less then or equal to the size of the allocated buffer. If the CPU is instructed to store a large quantity of data in the buffer (i.e., bigger then the size of the buffer), it will first fill the allocated buffer and the data that does not fit will be put into the next contiguous memory address. The contents of that address will be overwritten and when it is accessed an error will occur. The degree of severity of that error will depend on what was overwritten.

Parameter checking (or lack of parameter checking) is a very common problem in software that was written using the programming language "C" (as well as other lower-level languages). It is up to the developer to allocate enough memory to hold any data being passed to a subroutine. It is also the responsibility of the programmer to test the size of the parameter passed before copying it into the allocated buffer.

Large software projects are most often completed by several developers. The work is divided and in order for all the "pieces" to work together, a functional specification is created. The functional specification outlines exactly what parameters the routine should accept. In a commercial software effort, these "components" are often (though frequently not) unit tested. Unit testing is used to verify that the component works as specified. A part of the unit test is passing invalid parameters to a function and making sure the error is handled correctly.

OpenSSL

OpenSSL is a toolkit that allows a developer to implement SSL functionality in a variety of programming languages. As in any good toolkit, OpenSSL shields the developer from it's inner workings. OpenSSL is developed and maintained as "Open Source"[2]. There are a great number of people that work on the code. The developers work on a volunteer basis and there is no way to verify that all developers have extensively tested their code. The true strength in "open source" is that the source code is available to be viewed by anyone that wants to view it. The sheer number of people viewing the code greatly contributes to it's stability.

The Apache Server

The Apache web server is a robust open source implementation of web server. The Apache project was formed to create an open source HTTP server that would be available to anyone that wanted to use it as long as they agreed to the simple open source software licensing restrictions. Apache has grown to become the most used web server in the world. Because it is open source, the Apache code has been heavily scrutinized and is considered one of the most robust servers available.

[2] Open Source Initiative (OSI) http://www.opensource.org/

This paper describes vulnerabilities in certain versions of the Apache web server. The vulnerabilities described here are not necessarily in the actual Apache code. The security in the Apache server is provided through the inclusion of another open source project, mod_ssl. The mod_ssl project began with the stated goal of creating a simple “add-in” SSL module for the Apache server. It was envisioned that when SSL functionality is desired, a system manager would just install the mod_ssl component. After installation, the Apache server will have full SSL compatibility[3].

Origins of SSL

SSL was first used in the Netscape Navigator HTTP browser. SSL was added as a way to deal with the problem of transmitting confidential information using HTTP. This makes the web more secure for e-commerce where confidential financial information (such as credit card numbers) can be sent without the worry of interception. Transmitted HTML can be easily viewed by packet sniffers and SSL significantly reduces this risk. SSL makes even a basic access control/authentication scheme secure.

Although SSL is well defined, it is fairly difficult to implement. SSL requires complex cryptographic algorithms. Fortunately, theses algorithms are available in open source libraries. One of the first open source libraries was created by Eric Young. Mr. Young created a library that (when included in an application) provided SSL functionality. The library was called SSLeay. SSLeay later was used in the most widely accepted SSL toolkit, the OpenSSL toolkit. OpenSSL is a complete toolkit that implements all functionality that is needed to add SSL to an application.

As stated above, the popularity of the Apache server grew and there was a group of people that wanted to add SSL capabilities to the Apache server. An organization was created that’s purpose was to create an open source SSL module for the Apache server. mod_ssl is maintained by modssl.org to provide the Apache server with the desired functionality. mod_ssl uses the OpenSSL tool kit (which intern uses the SSLeay library) to achieve much of this functionality.

As it can be inferred by the previous paragraphs, the actual cause of vulnerabilities in Apache security is a very complex topic. Any “bug” that causes a vulnerability in the Apache server may be in the Apache source code, the mod_ssl code or the OpenSSL library. Or it can be in the interaction between the various components. If it has been determined that a vulnerability is in one of it’s support libraries, the implications can be far reaching. Many corporations base the SSL functionality of their own in-house SSL projects on OpenSSL. When an

[3] Modern versions of Apache come pre-installed with SSL functionality and mod_ssl (as a separate module) is no longer needed.

SSL vulnerability exists in Apache it is likely that it is also in many other applications that have used OpenSSL. . And that is the point of this paper.

Other things to consider

When a vulnerability is reported it is important to consider not only the vulnerability itself but the potentially far reaching consequences of the vulnerability. If a vulnerability is found in a base component, I (as a developer) must look at all projects that use the suspected component.

For example, I work for a large financial institution. Part of my job is to help other internal development groups implement security in their existing web-based applications. These applications have been written using a variety of programming languages such as C, C++, Java, J++ and Visual Basic. Because there is no universal toolkit for retrofitting SSL onto some of these languages. When a toolkit is unavailable, an SSL proxy or wrapper can be used.

A popular SSL wrapper is Stunnel. Stunnel can provide the required SSL functionality with a minimum of modification to the original application. Stunnel is an open source project and achieves it's SSL functionality by using the OpenSSL toolkit.

Thus far, there have not been any Stunnel specific exploits. Never the less, when investigating exploits in any tool kit, all applications that use the tool kit must be considered suspect.

The SSL Protocol
Introduction to SSL

The Secure Socket Layer protocol was developed by Netscape Corporation as a way to protect sensitive information (such as a credit card number) as it is sent over the internet. When a sniffer[4] is being used, packets that contain HTML can easily be understood. The risk of “credit card” theft is very high when using an unsecured channel. A hacker can easily extract a credit card number (from a web form) because it’s being sent in “clear text”.

The name of the HTTP protocol that has been encoded with SSL is HTTPS. When a browser connects with a server (typically port 443) over HTTPS, some iconic representation of a the secured session is usually displayed in the browser (i.e., a small lock). If a sniffer is being used, the contents of the session cannot be read.

Data is collected on web pages through the use of “forms”. The user fills in the form and clicks on the submit button (the button that tells the web browser to send the form to the host). Using the tool Ethereal[5] the captured HTML form looks as follows (Figure 1). Notice the “bolded” text. This is the contents of the HTML form. Anybody that is using a sniffer can easily read this information.

```
/twww/test/confirmation.asp HTTP/1.1
User-Agent: Mozilla/4.0 (compatible; MSIE 5.0; Windows 2000) Opera 6.01  [en]
Host: secure.mycompany.com
Accept: text/html, image/png, image/jpeg, image/gif, image/x-xbitmap, */*
Accept-Language: en
Accept-Charset: windows-1252;q=1.0, utf-8;q=1.0, utf-16;q=1.0, iso-8859-1;q=0.6, *;q=0.1
Accept-Encoding: deflate, gzip, x-gzip, identity, *;q=0
Referer: http://security.fmr.com/tbone/test/address.html
Cookie: SITESERVER=ID=cf5532350ff2f5ee74b4b836e19dc99b; theID=1328
Cookie2: $Version="1"
Connection: Keep-Alive, TE
TE: deflate, gzip, chunked, identity, trailers
Content-type: application/x-www-form-urlencoded
Content-length: 324
journalcode=XSC&subtype=freetrial&promotion=1405&title=Mr.
&christianname=Scott&surname=Smith&jobtitle=IT+Security+Consultant
&companyname=My+Company&addressline1=8000+Walter+ST.
&addressline2=&addressline3=Boston&addressline4=MA
&postcode=09878&telephonenumber=617+666666
&emailaddress=scott%40mycompany.com
```

Figure 1: Captured packet of an HTML transfer

[4] a “sniffer” is a tool that is used to capture data packets as they are sent across a network. Typically, the sniffer can decode these packets into a format that a person can read.

[5] Ethereal is an advanced packet sniffing tool. It can be down loaded from http://www.ethereal.com/

Background Information

There are a number of concepts that must be understood before the actual SSL protocol can be discussed. I will present them in the next few paragraphs as briefly as possible.

The strength of SSL is based on a component of Public Key Cryptography, the Digital Certificate. Although a detailed description of Public Key Cryptography is beyond the scope of this document, the following paragraphs should provide a basic understanding (which is required in order to understand the SSL protocol).

Speaking in the context of this paper, the are two types of encryption, Synchronous Cryptography and Asynchronous Cryptography. SSL uses both.

Synchronous Cryptography

Cryptographic algorithms use a "key" to encode text. A cryptographic algorithm could be visualized as a "black box". Clear text[6] goes into the box and Cipher Text comes out. A "key" is a string of characters that the cryptographic algorithm uses to "seed" the cryptographic process. Because each key is unique, the text coming out of the "black box" will be different for every key used. Conversely, the original key used to create the cipher text must be used to convert it back to the original clear text. This is the basis of synchronous cryptography (as can be seen in figure 2)

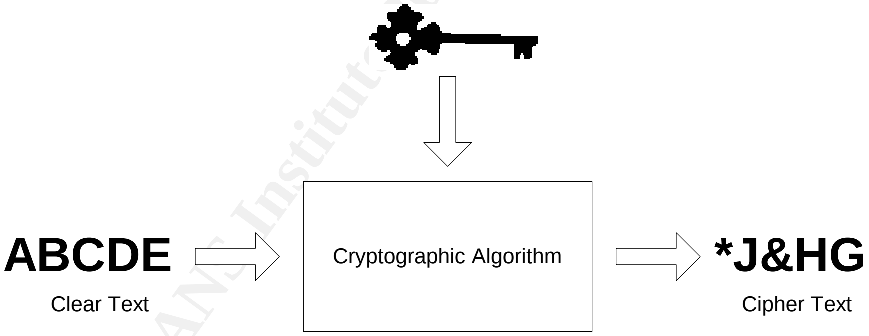


Figure 2: Encoding Clear Text

[6] "clear text" is the term for un-encoded text. Cipher text is the term for Clear Text that has been encoded by a cryptographic algorithm.

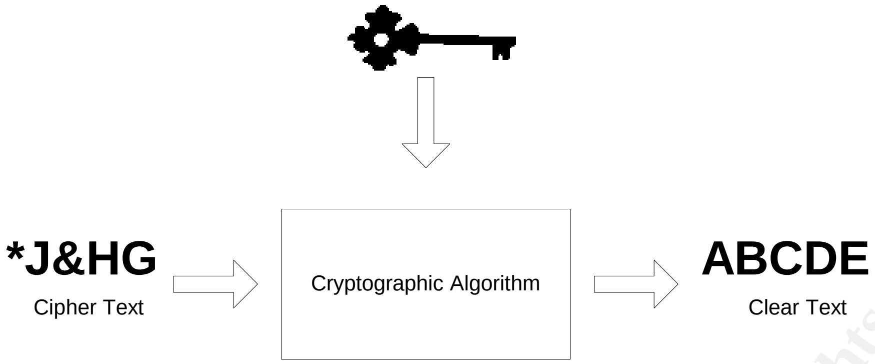


Figure 3: Decoding Cipher Text

Asynchronous Cryptography

Asynchronous cryptography takes a different approach. Two keys are used to encode and decode text. These keys are called “public” and “private” keys. If the sender wants to send confidential information to the receiver, the receiver’s public key is used (by the sender) to encode the cipher text. This is a “one way” transformation. The sender cannot decode the cipher text using the receiver’s public key (even though it was the receiver’s public key that was used to create the cipher text). Only the receiver’s private key can decode cipher text that was created using the receivers public key. Since the only instance of the receiver’s private key exists on the receivers computer, no one but the receiver can decode the cipher text.

This process works in reverse as well. If the sender encoded plain text using his own private key, only his public key could decode it. While this may not seem like a useful feature (since a user’s public key is available to the public) it is very useful when used for digital signatures. A user can “sign” an item of data by using his private key to encode the data. If the person at the receiving end can decode the data with the sender’s public key then the only way it could have been created was by the sender.

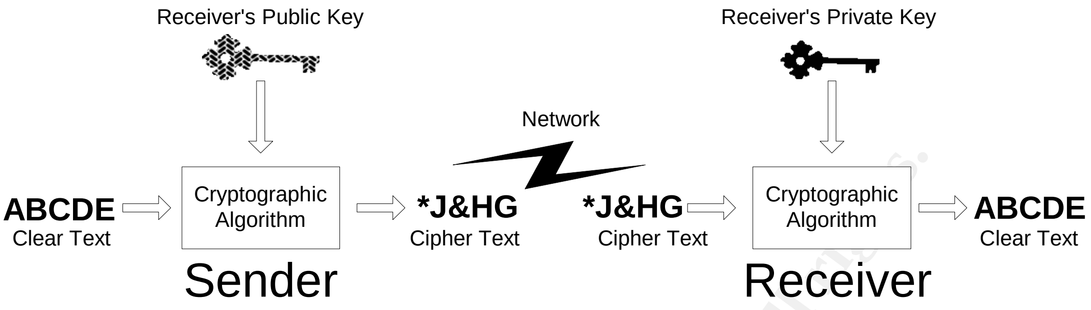


Figure 4: Asynchronous Encryption/Decryption

Digital Certificates

A Digital Certificate is a data structure that is used as a “container” to hold a user’s public key. The digital certificate actually contains a great deal more information that can be used by Public Key Infrastructures (PKI) but a description of this data is beyond the scope of this paper.

The most common way (though not the only way) to store a digital certificate is in X509 format. This format is widely accepted and there are numerous resources on X509 that can be accessed via the internet[7].

The Certificate Authority

There is one more concept that must be understood before presenting the SSL protocol. This is the concept of a Certificate Authority (CA). The CA is the software that actually creates the certificate. After creating a certificate, the CA “digitally signs it” with it’s own private key. As discussed above, the only way that this signature can be decoded is by using the CA’s public key.

During the SSL handshake, a server receives the certificate of the person that is trying to connect with it. A field containing the name of the CA (that issued the certificate) is extracted. The legitimacy of the certificate is determined by verifying that the public key (of the CA that issued the certificate) can correctly decode the certificate’s signature. At this point, SSL can be reasonably sure that the

[7] See the “Links” section in the appendix for specific URL’s.

certificate was created by a particular CA. Now that the origin of the certificate has been determined, the name of the CA is passed to a function that compares the CA name against a list of “trusted” Certificate Authorities. If it is trusted, then it is assumed that this certificate can be trusted (note that other fields are tested as well).

SSL In Action

The SSL protocol uses a combination of asymmetric and symmetric cryptography. Symmetric cryptography is much faster than asymmetric cryptography. Asymmetric cryptography provides better authentication techniques than is available using Symmetric Cryptography. An SSL session always begins with an exchange of messages called the SSL handshake. The handshake allows the server to authenticate itself to the client using Asymmetric Cryptography. Then it allows the client and the server to cooperate in the creation of symmetric keys used for rapid encryption, decryption, and tamper detection during the session that follows. Optionally, the handshake also allows the client to authenticate itself to the server.

The exact programmatic details of the messages exchanged during the SSL handshake are beyond the scope of this document. However, the steps involved can be summarized as follows.

1. The client sends the server the version number of the SSL it is using as well as other information (accepted cryptographic algorithms, randomly generated data, and other information the server needs to communicate with the client using SSL).

2. The server sends the client the server's SSL version number, cipher settings, randomly generated data, and other information the client needs to communicate with the server over SSL. The server also sends its own certificate and requests the client's certificate.

3. The client uses some of the information sent by the server to authenticate the server. If the server cannot be authenticated, the user is warned of the problem and informed that an encrypted and authenticated connection cannot be established. If the server can be successfully authenticated, the client goes on to Step 4.

4. Using all data generated in the handshake so far, the client (with the cooperation of the server, depending on the cipher being used) creates a series of bytes known as the pre-master secret. It then encrypts the pre-master secret with the server's public key (obtained from the server's certificate, sent in Step 2), and sends the encrypted pre-master secret to the server.

5. If the server has requested client authentication (an optional step in the handshake), the client also signs another piece of data that is unique to this handshake and known by both the client and server. In this case the client sends both the signed data and the client's own certificate to the server along with the encrypted premaster secret.

Before continuing with the session, Servers can be configured to check that the client's certificate is present in the user's entry in an LDAP directory. This configuration option provides one way of ensuring that the client's certificate has not been revoked.
It's important to note that both client and server authentication involve encrypting some piece of data with one key of a public-private key pair and decrypting it with the other key.

In the case of server authentication, the client encrypts the premaster secret with the server's public key. Only the corresponding private key can correctly decrypt the secret, so the client has some assurance that the identity associated with the public key is in fact the server with which the client is connected. Otherwise, the server cannot decrypt the premaster secret and cannot generate the symmetric keys required for the session, and the session will be terminated.

In the case of client authentication, the client encrypts some random data with the client's private key--that is, it creates a digital signature. The public key in the client's certificate can correctly validate the digital signature only if the corresponding private key was used. Otherwise, the server cannot validate the digital signature and the session is terminated.

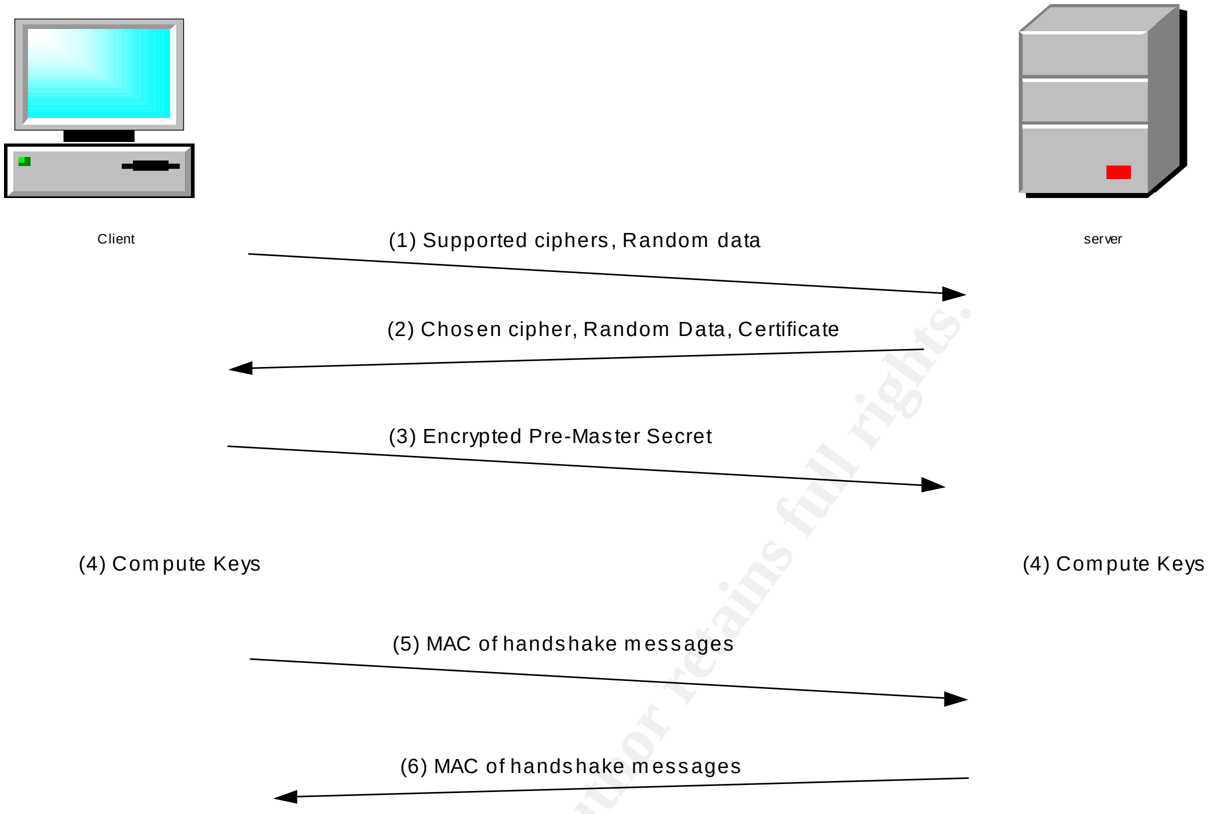

Client
server
(1) Supported ciphers, Random data
(2) Chosen cipher, Random Data, Certificate
(3) Encrypted Pre-Master Secret
(4) Compute Keys
(4) Compute Keys
(5) MAC of handshake messages
(6) MAC of handshake messages

Figure 5: Overview of SSL Handshake

Figure 6 is a greatly simplified version of the SSL handshake process. There are several variants of the SSL process that allow for varying degrees of security. When connecting to an online store, a browser will typically use a one-way SSL handshake. Since the user is going to be sending the store his credit card number, he would want to verify that the server has a trusted certificate.

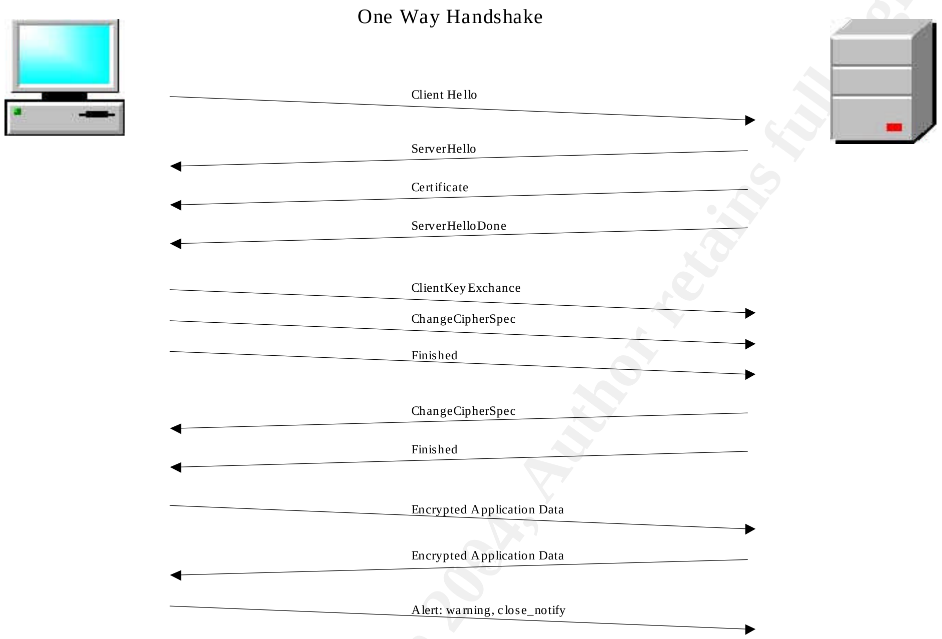
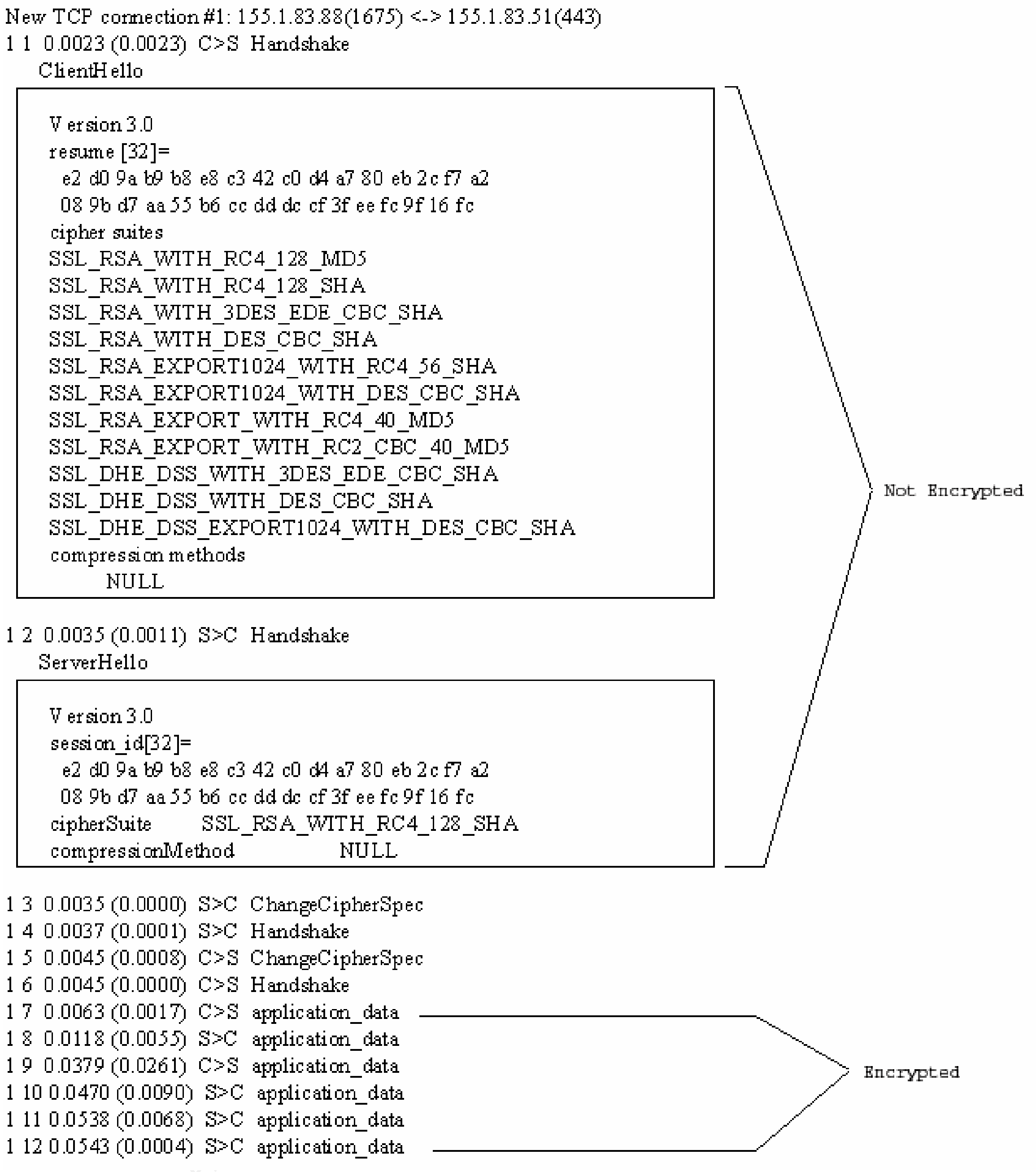


Figure 6: One way SSL handshake

This process can be “sniffed” by using a tool called ssldump. ssldump will display the various steps involved with the SSL handshake as they occur. ssldump is based on tcpdump. ssldump is useful for debugging handshake problems but cannot show the content of the packets once they are encoded (since it does not have access to the private key of the sender).



```
New TCP connection #1: 155.1.83.88(1675) <-> 155.1.83.51(443)
1 1  0.0023 (0.0023)  C>S  Handshake
     ClientHello
        Version 3.0
        resume [32]=
          e2 d0 9a b9 b8 e8 c3 42 c0 d4 a7 80 eb 2c f7 a2
          08 9b d7 aa 55 b6 cc dd dc cf 3f ee fc 9f 16 fc
        cipher suites
        SSL_RSA_WITH_RC4_128_MD5
        SSL_RSA_WITH_RC4_128_SHA
        SSL_RSA_WITH_3DES_EDE_CBC_SHA
        SSL_RSA_WITH_DES_CBC_SHA
        SSL_RSA_EXPORT1024_WITH_RC4_56_SHA
        SSL_RSA_EXPORT1024_WITH_DES_CBC_SHA
        SSL_RSA_EXPORT_WITH_RC4_40_MD5
        SSL_RSA_EXPORT_WITH_RC2_CBC_40_MD5
        SSL_DHE_DSS_WITH_3DES_EDE_CBC_SHA
        SSL_DHE_DSS_WITH_DES_CBC_SHA
        SSL_DHE_DSS_EXPORT1024_WITH_DES_CBC_SHA
        compression methods
                  NULL
1 2  0.0035 (0.0011)  S>C  Handshake
     ServerHello
        Version 3.0
        session_id[32]=
          e2 d0 9a b9 b8 e8 c3 42 c0 d4 a7 80 eb 2c f7 a2
          08 9b d7 aa 55 b6 cc dd dc cf 3f ee fc 9f 16 fc
        cipherSuite        SSL_RSA_WITH_RC4_128_SHA
        compressionMethod                 NULL
1 3  0.0035 (0.0000)  S>C  ChangeCipherSpec
1 4  0.0037 (0.0001)  S>C  Handshake
1 5  0.0045 (0.0008)  C>S  ChangeCipherSpec
1 6  0.0045 (0.0000)  C>S  Handshake
1 7  0.0063 (0.0017)  C>S  application_data
1 8  0.0118 (0.0055)  S>C  application_data
1 9  0.0379 (0.0261)  C>S  application_data
1 10 0.0470 (0.0090)  S>C  application_data
1 11 0.0538 (0.0068)  S>C  application_data
1 12 0.0543 (0.0004)  S>C  application_data
```

Figure 7: SSL handshake (using ssldump.exe[8])

SSL can run on any unused port. Typically it runs on port 443. This is the port commonly associated with HTTPS. If a user enters a URL beginning with "https://" the browser defaults to make a request to port 443 of the host. The host

[8] ssldump is a tool used to debug ssl sessions. It can be found at http://www.rtfm.com/ssldump/

can run SSL on any unassigned port but in order to connect with it, a user must specify a port number in his request.

SSL Vulnerabilities

The SSL protocol allows for a great number of options. There are not many (if any) vulnerabilities in the specification itself. The vulnerabilities that an application running SSL may experience are a direct result of the choices made by the developer who had implemented SSL.

When configuring SSL, most implementations use some kind of initialization file to determine which options are acceptable. One selectable option is which cryptographic algorithms can be used during the transfer. Different algorithms offer varying levels of security. Most cryptographic algorithms also allow for the use of varying key sizes. Essentially, the longer the key, the harder it is to “crack” the cipher text. Also, the longer the key, the more CPU processing is required to decode the cipher text. Selecting the minimum acceptable cipher is usually a trade-off between security and processing time.

Several cryptographic algorithms are often specified. During the handshake process, the client and the server search for an algorithm that is acceptable to both. Usually, the strongest algorithm (that is acceptable to both) is chosen. If both have listed a weak algorithm (such as an algorithm that uses a 40 bit key) it is possible that it will be selected during the handshake process. Cipher text created with a 40 bit key will be much easier to “crack” then using an algorithm that uses a larger key. Many companies have set the minimum acceptable algorithm as Triple DES (168 bits).

Depending on the server, there are some “vulnerabilities” that can be seen only when using cryptographic algorithms with larger keys. As stated above, the larger the key, the more CPU time is required to process SSL communications. On a high volume server, this can be a problem. Many companies agree on a compromise between the safety of a large key size and processing overhead. Using the strongest algorithms on an under-powered heavy volume server (which had worked fine serving HTML) could result in a denial of service situation. This is not really a vulnerability in SSL but one caused by the use of SSL.

Another SSL option that will result in vulnerabilities is when using a “null cipher”. As the name implies, a null cipher results in clear text being sent via SSL. A null cipher can be used if a server wants to establish a user’s identity through certificates but does not care if the information being sent is encoded. The “payload” of the message being sent through SSL (with a null cipher) can easily be “sniffed”. A developer may mistakenly allow a null cipher in order to “speed up” the SSL process. While it does require less processing cycles, if the data should be kept confidential using a NULL cipher would be a vulnerability of the system.

Other Potential Vulnerabilities

Compared with “cracking” a private key, a “man in the middle” attack is probably SSL’s easiest vulnerability to exploit. A “man in the middle” attack occurs during the handshake process.

1. A client sends a request to a server that he wants to establish an SSL session.
2. A rogue program responds to this request and sends the client its certificate.
3. An SSL session is established between the client and the rogue application
4. The rogue application sends a request to the server (that the client had originally intended to connect with) and requests a connection.
5. The server responds and sends it’s certificate.
6. A session is established between the rogue program and the server.

From then on, any client requests goes to the rogue program which decodes it using it’s private key. Looks at the contents (changes it if desired) and re-encodes it with the rogue computer’s private key. The packet is then sent to the server.

The process of actually inserting a rogue computer “in the middle” is actually much more complex then presented here. There are many ways to “intercept” the initial request from the host. Modifying a DNS or launching a DOS against the target host are two ways to do it

The best way to avoid a “man in the middle” situation is if both ends of the connection validate the supplied certificates. There are other “unorthodox” ways to do it as well. We (my company) have been experimenting with including extra information in a field of the certificate called it’s Distinguished Name. By including a computer’s ip address within the Distinguished Name a certificate can be created that would be bound to a particular machine. When the certificate is received, the ip address (of the computer that send the certificate) is compared with the ip address stored within the certificate. This adds an extra layer of protection from a man in the middle attack.

Another vulnerability exists in many internet-based applications that use SSL, particularly applications that may reside behind a company’s firewall. When designing applications that are to exist only behind a corporate firewall, security requirements are often “relaxed” in order to achieve greater speed. Many corporate applications use “one way SSL. In e-commerce, a typical “one way” SSL session is where the client verifies the server certificate but the server does not verify the client. In many internal applications that I have seen, the opposite is true. The server is interested in seeing the client’s (who in many cases is another server) certificate. The client is not concerned that it is speaking with the expected server. A developer may implement it this way because this is much easier to implement (since no extensive checking routines will be need to authenticate the server. If another application (running at the same company) was to pretend that it was the desired server, the imposter could receive information from the client application that is considered confidential. Obviously, this would require an employee of the same company to set up the application behind the corporate firewall, but it is possible.

As stated earlier, there are not many vulnerabilities in the SSL protocol itself. Where the biggest security risks occur is in the various SSL implementations. SSL is a complex standard to implement and it would make little sense to “reinvent the wheel” every time SSL is needed. The vast majority of developers will choose to use a proven implementation toolkit instead of creating one. This is where problems can occur since there could conceivably be undetected “bugs” in even a proven implementation.

A Specific Exploit

The vulnerabilities described in this paper have been documented in CA-2002-27

CA-2002-27	CERT® Advisory Apache/mod_ssl Worm	http://www.cert.org/advisories/CA-2002-27.html

The exploit being discussed here is listed as CA-2002-27 and is known as the Apache/mod_ssl worm. It is most commonly called "the Slapper Worm". Slapper takes advantage of a buffer overflow condition (described in VU#102795) that was originally found in mod_ssl. As mentioned above, mod_ssl was a component that was created in order to give the Apache server SSL capabilities. mod_ssl was based on OpenSSL.

Versions of OpenSSL that were released before version 0.9.6e and 0.9.7-beta 2 have a vulnerability that can result in a buffer overflow. If a client sends a very large key (i.e., bigger then the buffer that OpenSSL has reserved for it) the buffer will overflow.

When a software buffer overflows, the results are unpredictable. Before Object Oriented Programming, buffer overflows were the most common type of bug in application software. For example, a buffer was allocated to hold ten characters. At some point in the program, twelve characters are copied into the buffer. What ever was located in memory immediately beyond the allocated buffer will be "stepped on". If it was only a string, the result would be a malformed string every time that string was displayed. If the two character over-run stepped on program code, it could have worse consequences.

Buffer overflows in internet based applications can allow a hacker to cause varying amounts of damage. It is the hacker's intent is to cause a denial of service (DOS) on the machine being attacked, a buffer overflow is easy to exploit. The hacker merely passes a very large amount of data into a program that was expecting a much smaller amount of data. If the program does not test the size of the supplied data before moving it into the allocated buffer (and compare it with the size of the allocated buffer) the program may crash and no other client will be able to access it.

A more complex exploit would be to overflow the buffer and insert actual machine code. The hacker could use varying numbers of noops[9] to fill the server stack until a point of execution is found. At that point the machine language commands are inserted and executed. From that point, the attacker could instruct the computer to do just about anything. That is what the Slapper Worm does. It causes a buffer overflow then inserts it's own executable code.

[9] a "noop" (pronounced no op) is an assembly language instruction that essentially does nothing. It is used to pad boundaries that are specific to particular systems.

The following systems have been tested for this vulnerability[10]

Systems Affected

Vendor	Status	Date Updated
Apache-SSL	Unknown	9-Aug-2002
Apple Computer Inc	Vulnerable	9-Aug-2002
Covalent	Unknown	9-Aug-2002
Debian	Vulnerable	9-Aug-2002
Gentoo Linux	Vulnerable	9-Aug-2002
Guardian Digital	Vulnerable	9-Aug-2002
Hewlett-Packard Company	Vulnerable	9-Aug-2002
IBM	Vulnerable	9-Aug-2002
Juniper Networks	Vulnerable	16-Aug-2002
Lotus Development Corporation	Not Vulnerable	9-Aug-2002
NCSA	Unknown	9-Aug-2002
NetBSD	Vulnerable	9-Aug-2002
OpenLDAP	Vulnerable	9-Aug-2002
OpenPKG	Vulnerable	9-Aug-2002
OpenSSL	Vulnerable	30-Jul-2002
Oracle	Vulnerable	9-Aug-2002
Red Hat Inc.	Vulnerable	9-Aug-2002
RSA Security	Vulnerable	13-Sep-2002
Trustix	Vulnerable	9-Aug-2002

Table 1: Effected Systems

Apache/mod_ssl Worm

Slapper

The Slapper worm takes advantage of the afore-mentioned buffer overflow. The virus writer has determined at exactly what point the buffer will overflow. Then instructions are entered that will force the victim to execute a number of tasks. This particular worm was designed specifically to run on Linux systems. It assumes that other common programs have been installed (on the victims system) and uses these programs to perform various tasks. Since the exploit allows for many possibilities, the original worm code has already been used to create multiple variations of it self.

The Slapper Worm works as follows.

1. First, it scans port 80/tcp using an invalid HTTP GET request. The request looks as follows: GET / HTTP/1.1

[10] From VU#102795

Using netcat[11] this request was sent to various servers. Table 2 shows the results. Note that the Slapper Worm is a Linux specific worm. Windows systems are present here for comparative purposes.

OS	Result
Red Hat 6.2	HTTP/1.1 400 Bad Request Date: Fri, 27 Sep 2002 11:58:57 GMT Server: Apache/1.3.12 (Unix) (Red Hat/Linux) PHP/3.0.15 mod_perl/1.21 Connection: close Transfer-Encoding: chunked Content-Type: text/html; charset=iso-8859-1
Red Hat 7.2	HTTP/1.1 400 Bad Request Date: Fri, 27 Sep 2002 09:11:59 GMT Server: Apache/1.3.20 (Unix) (Red-Hat/Linux) Connection: close Transfer-Encoding: chunked Content-Type: text/html; charset=iso-8859-1
Mandrake 7.2	HTTP/1.1 400 Bad Request Date: Fri, 27 Sep 2002 14:24:47 GMT Server: Apache-AdvancedExtranetServer/1.3.14 (Linux-Mandrake/2mdk) mod_ssl/2.7.1 OpenSSL/0.9.5a PHP/4.0.3pl1 ApacheJServ/1.1.2 Connection: close Transfer-Encoding: chunked Content-Type: text/html; charset=iso-8859-1
Windows NT	HTTP/1.0 200 OK Server: Microsoft-IIS/2.0 Date: Fri, 27 Sep 2002 15:07:40 GMT Content-Type: text/html Accept-Ranges: bytes Last-Modified: Mon, 18 Nov 1996 01:38:10 GMT Content-Length: 4051
Windows 2000	HTTP/1.1 400 Bad Request Server: Microsoft-IIS/5.0 Date: Fri, 27 Sep 2002 15:16:22 GMT Connection: close Content-Length: 3212 Content-Type: text/html

Table 2: Results of sending the GET / HTTP/1.1 request using netcat

2. Slapper tests thatApache is running with the following line:

```
if (strncmp(a,"Apache",6)) exit(0);
```

[11] using host ip and port 80 (ex. /nc 192.168.1.1 80)

3. Slapper then compares data extracted from step 1 with a list of potentially vulnerable systems (as can be seen in Figure 6).

```
struct archs {
        char *os;
        char *apache;
        int func_addr;
} architectures[] = {
        {"Gentoo", "", 0x08086c34},
        {"Debian", "1.3.26", 0x080863cc},
        {"Red-Hat", "1.3.6", 0x080707ec},
        {"Red-Hat", "1.3.9", 0x0808ccc4},
        {"Red-Hat", "1.3.12", 0x0808f614},
        {"Red-Hat", "1.3.12", 0x0809251c},
        {"Red-Hat", "1.3.19", 0x0809af8c},
        {"Red-Hat", "1.3.20", 0x080994d4},
        {"Red-Hat", "1.3.26", 0x08161c14},
        {"Red-Hat", "1.3.23", 0x0808528c},
        {"Red-Hat", "1.3.22", 0x0808400c},
        {"SuSE", "1.3.12", 0x0809f54c},
        {"SuSE", "1.3.17", 0x08099984},
        {"SuSE", "1.3.19", 0x08099ec8},
        {"SuSE", "1.3.20", 0x08099da8},
        {"SuSE", "1.3.23", 0x08086168},
        {"SuSE", "1.3.23", 0x080861c8},
        {"Mandrake", "1.3.14", 0x0809d6c4},
        {"Mandrake", "1.3.19", 0x0809ea98},
        {"Mandrake", "1.3.20", 0x0809e97c},
        {"Mandrake", "1.3.23", 0x08086580},
        {"Slackware", "1.3.26", 0x083d37fc},
        {"Slackware", "1.3.26",0x080b2100}
};
```

Figure 8: Data structure used in the actual Slapper Worm

4. After it has been established that a system is vulnerable, an SSL handshake is begun.

5. Slapper performs the buffer-overflow by supplying the following data structure in the hand-shake.

```
unsigned char overwrite_session_id_length[] =
        "AAAA"
        "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
        "\x70\x00\x00\x00";
```

6.Slapper continues to over-write the buffer with it’s own code. It uses the parameters in Figure 8 to determine system specific information.

7. The file /tmp/uubugtraq is created.

8. uubugtraq is decoded to /tmp/bugtraq.c.
9. bugtraq.c is then locally compiled to bugtraq.

10. bugtraq is then run

What is does from that point depends on the variant of Slapper. Slapper (A) sets up a peer to peer network. The per-to-peer net work can be used to launch a distributed denial of service attack, send files or just about anything the hacker can code.

In Summary, the Slapper Worm really doesn't do anything unique. It's author takes advantage of a known buffer overflow situation. After exploiting the vulnerability, it can do a number of different things.

As discussed at the beginning of this paper, this type of vulnerability can be easily avoided. If the programmer had made certain that he checked the size of all parameters being passed, this exploit would not have been possible.

Protection

If running a system that is vulnerable to this attack, the best means of protection is patching the system with the latest mod_ssl. If possible, upgrading to the newest Apache would also resolve this problem.

If there is a reason why the aforementioned upgrades are not possible, There is a "kludge" that can be done that will make this attack impossible. Login as "root". Create a text file named /tmp/uubugtrq and /tmp/bugtraq.c (to be safe) and set it to "read only". This will make it impossible for Slapper to create and compile the required file. Note that this is not a good solution to the problem but will guarantee that your machine does not take part in a distributed denial of service against another machine. If the mod_ssl code is not upgraded, the buffer overflow could cause your system to become unstable (which will could result in a denial of service for anyone trying to access you server).

It should also be noted that if a system does not have a "C" compiler, the Slapper Worm will not be able to compile and run itself. Even if a "C" compiler is not on a system, the vulnerable system should still be patched for the reasons discussed above.

Removal

It can be determined if a system is infected by looking in the /tmp directory for the three files discussed above (uubugtraq, bugtraq.c, bugtraq). If these files are detected, deleting them and rebooting the system will correct the problem. Upon re-booting, an error message will probably be returned (since the system will try to run bugtraq code) until it's entry has been removed.

Variants

Slapper B

Slapper B exploits the same vulnerability as Slapper (A). After executing the buffer overload, it creates a the file /tmp/.cinik.uu. This file is decoded to /tmp/.cinik.c. The gcc compiler is then run and the result is the executable /tmp/.cinik

This variant creates a script (/tmp/.cinik.go) which is used to collect system configuration information. This information is sent to an email address (probably the virus writer).

The worm also adds itself to the crontab file causing the file to be restarted hourly (in case it has been terminated).

Slapper .B effects the following Linux distributions:

- Debian Linux running HTTP Server 1.3.26
- Red Hat Linux running HTTP Server 1.3.6, 1.3.9, 1.3.12, 1.3.19, 1.3.20 or 1.3.23
- SuSE Linux running HTTP Server 1.3.12, 1.3.17, 1.3.19, 1.3.20 or 1.3.23
- Linux-Mandrake running HTTP Server 1.3.14, 1.3.19, 1.3.20 or 1.3.23
- Slackware Linux running HTTP Server 1.3.26

Slapper C

Slapper C is a modified variant of Slapper.A and exploits the same vulnerability as Slapper (A). Slapper C uses a port 4156 instead of port 2002. The file names are different as well.

Slapper C uploads itself as /tmp/.unlock.uu and decodes the file to /tmp/.unlock. It uses tar to decompress its content. Slapper C is also compiled by the gcc compiler. Slapper next compiles the extracted source code /tmp/.unlock.c to /tmp/httpd, starts it and removes all files except /tmp/.unlock.

This variant also sends IP addresses of infected hosts via email probably to the virus writer.

The worm also adds itself to the crontab file causing the file to be restarted hourly (in case it has been terminated).

Effected Distributions:

- Debian Linux running HTTP Server 1.3.26

- Red Hat Linux running HTTP Server 1.3.6, 1.3.9, 1.3.12, 1.3.19, 1.3.20 or 1.3.23
- SuSE Linux running HTTP Server 1.3.12, 1.3.17, 1.3.19, 1.3.20 or 1.3.23
- Linux-Mandrake running HTTP Server 1.3.14, 1.3.19, 1.3.20 or 1.3.23
- Slackware Linux running HTTP Server 1.3.26

Conclusions from a Programmatic Viewpoint

The lesson that can be learned from Slapper is that when developing a software component, the developer must not assume that the user of the component will use it properly or with due diligence. The developer that is using the component should not assume that the component is “rock solid” and must try to protect it from his own code. His code may become a component in a larger system where it’s caller may not be as competent as he is. Thus, he must assume that a future user of his code may make mistakes (or assumptions about his code) that could lead to a security risk.

Code being written for the Internet can be accessed by a limitless number of anonymous users. It should not be assumed that all these users have good intensions. It is unlikely that the particular vulnerability being exploited by Slapper would ever be detected in normal use (i.e., if a hacker never intentionally tried to exploit it). Developers of Internet code must assume that their code is running in a hostile environment.

Traditionally, the best practice for a software developer is to reuse components that have been previously created (as opposed to starting over from scratch). In a mission critical application, the components being considered for inclusion must be carefully evaluated as to their exact inherent risk. If a component proves to be unstable it should not be used. It is probably a good idea to test the stability of a component (as in unit testing) rather then relying on the assumption that since it is used in other applications, it must be fine.

Please note, I am not criticizing the OpenSSL development team. OpenSSL is a great tool and I recommend it highly. The “heart” of OpenSSL (the SSLeay Libray), and the early versions of OpenSSL (that were used in the mod_ssl component) as well as mod_ssl itself, were written at a time when the threat of hackers was not on top of a developers worry list. mod_ssl was created as way to introduce security into Apache web based applications. The threat being addressed was the disclosure of private information. It is unlikely that (at that time) anyone could have expected that solution would create an entirely different problem.

Bibliography

Understanding Public-Key Infrastructure, Concepts, Standards, and Deployment Considerations
Carlisle Adams, Steve Lloyd
Copyright ©1999 Macmillan Technical Publishing

Digital Certificates, Applied Internet Security
Jalal Feghhi, Jalil Feghhi, Peter Williams
Copyright ©1999 by Addison-Wesley Longman, Inc.

TCP/IP Illustrated Volume 1, The Protocols
W. Richard Stevens
Copyright ©1994 by Addison-Wesley

SSL and TLS, Designing and Building Secure Systems.
Eric Rescorla
Copyright ©2001 by Addison-Wesley

Network Security with OpenSSL
John Viega, Matt Messier & Pravir Chandra
Copyright © 2002 O'Reilly & Associates Inc

Links

OpenSSL Vulnerability

CVE	http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0656
CERT® Advisory CA-2002-23 Multiple Vulnerabilities In OpenSSL	http://www.cert.org/advisories/CA-2002-23.html
Vulnerability Note VU#102795	http://www.kb.cert.org/vuls/id/102795
National Infrastructure Proctection Center	http://www.nipc.gov/warnings/advisories/2002/02-006.htm
E-SECURE-DB IT Security Information DATABASE	http://www.e-secure-db.us/dscgi/ds.py/View/Collection-348

mod_ssl Exploit

CERT® Advisory CA-2002-27 Apache/mod_ssl Worm	http://www.cert.org/advisories/CA-2002-27.html
F-Secure Virus Descriptions	http://www.f-secure.com/v-descs/slapper.shtml
Global Slapper Worm Information Center	http://www.f-secure.com/slapper/

Internet Storm Center	http://isc.incidents.org/analysis.html?id=167
Internet Security Systems Security Alert	https://gtoc.iss.net/issEn/delivery/xforce/alertdetail.jsp?oid=21130
McAffe Virus Information	http://vil.mcafee.com/dispVirus.asp?virus_k=99693
Symantec	http://securityresponse.symantec.com/avcenter/venc/data/linux.slapper.worm.html

Misc. Articles

Linux server worm exploits known flaw	http://news.com.com/2100-1001-957987.html?tag=rn
Linux worm causes peer pressure	http://news.com.com/2100-1001-958122.html?tag=fd_top
Linux worm creating P2P attack network	http://news.com.com/2100-1001-957988.html?tag=rn
Arrest for Slapper author	http://www.vnunet.com/News/1135274
Slapper worm spreads its disease	http://www.vnunet.com/News/1135137
Third slapper worm hits the street	http://www.vnunet.com/News/1135304

Organizations

Open Source	http://www.opensource.org/
mod_ssl	http://www.modssl.org/
The Apace Software foundation	http://www.apache.org/
Open SSL Project	http://www.openssl.org/

Tools

ssldump	http://www.rtfm.com/ssldump/
tcpdump	http://www.tcpdump.org/
Ethereal	http://www.ethereal.com/