Lessons from Heartbleed

Chanathip Namprempre

Department of Electrical and Computer Engineering, Faculty of Engineering,

Thammasat University, Rangsit Campus, Pathum Thani 12121

Abstract

Heartbleed is a major security flaw affecting many computer systems. It is easy to mount and leaves few traces. It allows an attacker to extract a portion of the memory contents of the targeted computer, which could be either a server or a client. Cryptography offers many tools that could have been used to mitigate the negative repercussions of this breach. Unfortunately, evidence suggests that not many of these tools were used by servers at the time of the announcement of Heartbleed. This paper describes what these tools are and how they could have helped. The goal is to raise awareness so that users and system administrators alike can understand the benefits that existing, well-known cryptographic tools have to offer to cope with practical attacks.

Keywords: public key cryptography, forward security, password-based cryptography.

1. Introduction

Heartbleed is a major security flaw affecting many computer systems. Its existence was announced in April 2014. Since then, it has ignited much interest among diverse groups of people, from server administrators, cryptographers, and programmers to common users and policy makers.

Heartbleed is easy to mount and leaves few traces. It allows an attacker to extract a portion of the memory contents of the targeted computer, which could be either a server or a client. The memory contents vulnerable to exposure could include many things such as the long-lived secret key of the server, the usernames and passwords of the users, and other information that may be contained in the payload being exchanged during the session.

The repercussions from the exposure of sensitive information are often costly to address. It would have been better if more had been done to limit the scope of damage in the event that secrets are exposed.

Cryptography offers many tools that could have been used to mitigate the negative repercussions of this breach. Unfortunately, evidence suggests that not many of these tools were used by servers at the time of the announcement of the attack. This paper describes what these tools are and how they could have helped improve the situation. The goal is to raise awareness so that users and system administrators alike can see and understand the benefits that existing, wellknown cryptographic tools have to offer to cope with practical attacks.

2. Background

Heartbleed exploits a flaw in OpenSSL, an implementation of the SSL/TLS protocol. In this section, we first discuss in general terms the need for and the basic ideas behind the SSL/TLS protocol.

2.1 Public key cryptography and the secure channel

Consider a typical interaction between a client and a server. The client is often a web browser operated by a user who would like to access resources on the server. The server ensures that an incoming request for resources indeed comes from a legitimate user by asking the user to type in his or her username and the corresponding password, which have presumably been registered with the server prior to the current communication session. Once the server verifies that the username and password are correct, it allows the client to access the resources the user is authorized for.

We should note that even though this method is neither the only nor the best way to authenticate communicating parties and to protect the ensuing exchanges of information between them, it is by far the most popular and will be the focus of this paper.

THE KEY-THEN-DATA-EXCHANGE APPROACH. Since the username-password pair that a user sends to a server is the *information advantage* that distinguishes a legitimate user from someone without authorization for the requested resources, it should not be sent over the public network in the clear, lest it be easily captured by packet sniffers. A *secure channel* for transmission is needed. For the purposes of this paper, we can conceptually regard a secure channel as a pipe between two parties in which data being exchanged through the pipe are encrypted and authenticated. (It has long been accepted that "encryption without integrity checking is all but useless" [6].)

Given that symmetric-key cryptosystems are much more efficient than public-key ones, the bulk of the data exchange is almost always implemented with symmetric-key cryptographic primitives. However, requiring that every pair of client and server on the Internet possesses a shared secret key (a bitstring known only to the two of them and none others) before they can securely communicate is clearly impractical.

In practice, public-key cryptography is used to address this problem. A party holds a pair of public and secret keys which are generated in such a way that

– if the public key is used to encrypt a message to obtain a ciphertext, the secret key can be used to decrypt the ciphertext, and
– if the secret key is used to sign a message, the public key can be used to verify the signature.

There can be many other properties depending on the cryptosystems in question, but these two are the most well-known. For the purpose of building a secure channel between two parties, public-key cryptography is used to obtain a shared secret key between the parties, a process known as *key exchange* (KE). Bootstrapping from the public-secret key pairs of the participants, a key exchange protocol allows the two parties to arrive at a shared secret known only to them even if an adversary may, at the very least, observe the entire sequence of exchanges during the protocol's execution.

Currently, the most common kind of key exchange used in practice authenticates only one of the participants, namely the server. Specifically, after completing the key exchange protocol with a server, the client can be *somewhat* confident that the server is who it claims to be while the server has no idea who the client is. In most cases, the identity of the client is ascertained after the key exchange protocol completes via the submission of an appropriate username- password pair during the data exchange.

The level of confidence that the client can have about the identity of the server after the completion of the key exchange protocol depends on the quality of the *certificate* that the server uses. Essentially, a certificate is a bundle of data about its owner. Its main role is to bind the owner's public key and its name together. Other relevant information is included, for example, the expiration date of the certificate and the issuer's name. The binding is implemented via digital signatures. Specifically, the authenticity of a certificate is asserted by its issuer by having the latter sign it. A client who is in possession of a server's certificate can check whether the certificate is trustworthy by verifying the accompanying digital signature under the public key of the issuer, thus creating a chain of trust: the client who starts out trusting the authenticity of the issuer's public key can now trust the authenticity of the server's public key.

Therefore, the certificate of an entity is only as trustworthy as the certificate of its issuer. Two important consequences of this trust dependency are the following.

– Once the secret key corresponding to the public key contained in the certificate of the issuer is exposed, none of the certificates signed by the issuer after the exposure can be trusted.

– Care must be taken when a *self-signed certificate* is used. A self-signed certificate is one for which the issuer and the certificate owner are the same entity. Hence, in effect, a self-signed certificate that asserts that a particular public key *pk* truly belongs to an entity *A* says that we should believe that *A* owns *pk* simply because *A* says so. Clearly, asking for an entity to vouch for its own trusthworthiness is in general not a good idea.

  We remark that scenarios in which self-signed certificates are useful exist. (For example, an administrator in an enterprise network can manually install the self-signed certificate of a server in the enterprise on the employees' computers so as to allow client programs to be able to verify the authenticity of the certificate.) Nonetheless, in general, self- signed certificates should be employed carefully and minimally or be avoided altogether.

We emphasize that trusting that a public key belongs to an entity simply means that we believe that the owner of the public key knows the secret key corresponding to the public key. In a system in which a break-in has not occurred, this means that the *only* entity that knows the secret key should be the owner of the matching public key. In contrast, in the event of a breach, the assertion that the public key truly belongs to its owner becomes beside the point because an adversary may now also have the secret key. The information advantage that previously had distinguished the legitimate owner of the public key from the adversary has vanished.

PROTECTING WEB COMMUNICATION VIA SSL/TLS. The Transport Layer Security (TLS) protocol is the successor of the Secure Socket Layer (SSL) protocol. Most web servers that support secure communication (that is, in simple terms, ones that allow clients to connect to them with the prefix https://...) support both protocols. We do not distinguish between the two in our discussion in this paper and simply refer to the SSL and the TLS protocols collectively as SSL/TLS.

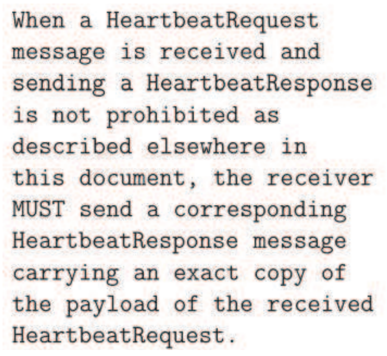
Consider the interaction between a client and a server using the key-then-data-exchange approach discussed above. When the interaction is performed via the SSL/TLS protocol, the key exchange part corresponds to the *handshake protocol* while the data exchange part corresponds to the *record protocol*. As in any real implementation of a cryptographic protocol, there are many tasks to be performed other than the main ones that the protocol is designed to accomplish. For example, in this case, the main tasks are key exchange and data exchange, but SSL/TLS must also allow the participants to negotiate which cryptographic algorithms and protocols (and which of the available versions) they would like to use for the key negotiation (e.g. Diffie-Hellman, Elliptic Curve Diffie-Hellman, etc.) and data exchange (AES CBC, AES GCM, etc.).

3. The Heartbleed Attack

There are many implementations of SSL/TLS. One of the most popular implementation is the open source library called OpenSSL [22]. The Heartbleed breach is due to a bug, which we will refer to simply as the Heartbleed bug, in widely-adopted versions (namely, versions 1.0.1 through 1.0.2beta) of this library [23].

THE CAUSE. The Heartbleed bug was borne out of the need to prevent the connection between a client and a server from getting prematurely torn down during a session. Although SSL/TLS operates over the Transmission Control Protocol (TCP), which is connection-oriented, and thus needs not be concerned with this problem, the Datagram Transport Layer Security (DTLS) protocol, whose goal is to achieve similar security guarantees as SSL/TLS over datagram protocols, does. In response to this need, the *Heartbeat Extension* was proposed in RFC 6520 as a protocol for keeping connections alive for both DTLS and TLS [18]. As it happened, in an attempt to check for the "heartbeats" of the participants during the session so as to keep the current connection alive, an OpenSSL developer inadvertently introduced the Heartbleed bug [17].

THE HEARTBLEED BUG. Suppose a client sends a server a Heartbeat Request and expects a Heartbeat Response to be returned. RFC 6520 describes what must happen.



```
When a HeartbeatRequest
message is received and
sending a HeartbeatResponse
is not prohibited as
described elsewhere in
this document, the receiver
MUST send a corresponding
HeartbeatResponse message
carrying an exact copy of
the payload of the received
HeartbeatRequest.
```

In a normal situation, the client would send the server a payload (e.g. a string "hello") and the corresponding size (e.g. 5 bytes). (A fixed-size, random padding is also sent, but this padding is inconsequential with respect to the Heartbleed bug.) In an attack, however, the adversary sends a Heart-beatRequest that contains a small payload along with a purported size that is larger than the actual size of the payload. This attack works because the code that deals with the HeartbeatRequest simply trusts that the purported size is correct and du- tifully copies and returns the data in its memory, starting at the pointer to the received payload, for the amount specified by the purported size which can be as large as 64K bytes [18]. Thus, when an attacker sends a small payload and a large purported size, it gets back not only the payload it has sent but also the data that reside in the server's memory immediately after the payload ends, for the number of bytes specified by the size parameter. It is the extraneous data returned here that is the crux of the vulnerability. For example, it has been demonstrated that the targeted server's long-term secret key can be obtained through this attack [21].

Not surprisingly, attacks exploiting the Heartbleed bug are effective against both servers and clients. Some have termed a Heartbleed attack against clients *Reverse Heartbleed* [2]. In particular, an attacker can set up a malicious server and trick a user to establish an SSL/TLS connection to it. Once the connection is established, the server can send a malicious Heartbeat-Request to the client and extract a portion of the client's memory contents. In effect, this variation of the Heartbleed attack adds power to garden-variety phishing scams.

Although Reverse Heartbleed is a simple variation of Heartbleed, its attack surface is much larger than that of Heartbleed for many reasons. First, clients can run on many platforms, ranging from home computers and mobile devices to embedded systems. This means that patching clients is much more challenging compared to patching servers. Second, client systems are often less well-maintained, and maintenance is often done by end users. This means that vulnerabilities may be undetected or neglected for years. Considering the prevalent use of computing devices in this day and age, Reverse Heartbleed may end up doing more damage over a longer period of time than Heartbleed itself.

4. Repercussions of Heartbleed

Once an adversary is able to get a portion of the memory contents of the target, the amount and the kind of damage it can inflict can be large and varied. Mitigation and prevention depend on what kind of information the adversary has obtained from the memory.

A LONG-TERM SECRET KEY IS STOLEN FROM MEMORY. One of the most feared situations is that adversary *A* has obtained the server's long-term secret key sk_s from the server's memory. Clearly, any server that has been afflicted by the Heartbleed bug must, at the very least, change its longterm secret key. In this section, we discuss additional problems and approaches to mitigation that also need to take place.

Breaking KE. The long-term secret key is used in the key exchange protocol, in which the server's identity is authenticated and a session key K is established. If A has captured the protocol flows during the key exchange process, then an adversary may be able to use sk_s to obtain K. Armed with K, the adversary can decrypt any ciphertext that it has captured in the past for this session. A cryptographic tool that, if it had been used from the beginning, could have prevented A from obtaining K is called a *forward-secure key exchange protocol*. This type of protocol is described in Section 5.1.

Forging certificates. Once armed with sk_s, the adversary A can sign anything it would like to on behalf of the server. Consequently, any signature purported to be that of the server should no longer be trusted as having been generated by the server. Unfortunately, this includes any certificate that the server has *already* issued. Therefore, any certificate in the chain downstream from that issued by the server whose secret key has been compromised must be *revoked*, an undesirable process that incurs a large overhead. A cryptographic tool that could have eliminated the need for certificate revocation is a *forward-secure signature scheme.* This type of scheme is described in Section 5.2.

Forging signatures on documents. The authenticity of documents that the server has signed so far using the compromised sk_s can no longer be trusted. As with certificates, a tool to address this problem is a forward-secure signature scheme.

We should note that although the problems mentioned throughout this section focus on what happens to servers, many are also applicable to clients. It bears repeating that even users that do not run servers may be affected by the Heartbleed bug. Through a Reverse Heartbleed attack, a user who has been tricked into making a SSL/TLS connection to a malicious server mounting the attack could end up with his or her long-term secret key stolen. Any application that uses this compromised secret key would be affected. For example, if the secret key is also used for signing digital documents, all previously generated signatures would no longer be trustworthy. Forward- secure signatures can help address this particular problem.

CREDENTIALS ARE STOLEN FROM MEMORY. If an adversary can obtain certain credentials directly from memory, the damage depends on how the credentials are to be used. We discuss the two most common types of credentials here.

Username and password. Although this information allows the adversary to log in as a legitimate user to the targeted services, in practice the damage resulting from the exposure of a user's password is often not limited to only the services currently under attack. The reason is that many users reuse their passwords for many different services, consequently rendering those services vulnerable also. A cryptographic tool that could have helped prevent this is *password-based authenticated key exchange* described in Section 5.4.

Credit card number and other info. If an adversary can get all of the information necessary to charge a credit card in an online transaction (in general, this includes the credit card number, the last three digits of the sequence of numbers on the back of the card, and the expiration date), then it can charge any purchases to the card at will, until the card is cancelled. A cryptographic tool that could have helped prevent this is *limited-use credit card numbers*. Such numbers are described in Section 5.5.

A PSEUDORANDOM NUMBER GENERATOR SEED IS STOLEN FROM MEMORY. Most programs that implement secure functionality need to use randomness. One of the most common ways to programmatically generate randomness is to use a pseudorandom bit generator (PRG). This primitive requires that a *seed*, a short sequence of random bits, is given as an input so that a longer sequence of pseudorandom bits can be *deterministically* computed and output. If the output is computed deterministically without using any secrets (such as hidden state or a secret key), then anyone in possession of the right seed can easily compute the output.

If the stolen seed had been used to generate a session key, then the adversary can use the seed to compute the session key then decrypt any ciphertext it has collected for this session. A cryptographic tool that could have addressed this is a *forward- secure pseudorandom bit generator*. Such generators are described in Section 5.3.

A SESSION TICKET IS STOLEN FROM MEMORY. A *session ticket key* is used to protect a *session ticket* containing state information about the current connection so that the server can resume a SSL/TLS session without performing the full handshake, thus reducing the server load. If a session ticket key is stolen, then an adversary can (1) decrypt previously-issued session tickets and (2) generate new session tickets. For the first threat, since session tickets contain state information about a session, the potential damage depends on what the server decides to include in a ticket. The second threat, however, is unlikely to be much of an issue since session ticket keys are often freshly generated at server reboots.

Clearly, there are other types of information that, once stolen from memory, can cause damage to the victim. We only consider the ones mentioned above since they have the most damaging repercussions.

A CAVEAT. Although there are many types of sensitive data that, once stolen, can lead to extensive damage as discussed above, we emphasize here that, in general, it is not a simple matter for an attacker to successfully pick and choose the type of data to steal at will. In fact, when the Heart- bleed bug was first discovered, the initial impression among security experts was that a server's long-term secret key could not be stolen through Heartbleed, although, as it happened, this impression was quickly proven to be false [20]. Given the level of sophistication required of an adversary to ensure that the desired data end up residing in the appropriate region of the memory in order to mount the attack, it would be arguably much simpler and more fruitful for an attacker to instead resort to social engineering attacks such as phishing.

5. Prevention through Cryptography

In this section, we discuss what could have been: what kinds of damage could be mitigated if certain types of cryptosystems had been deployed. Although it is no secret that these cryptosystems exist and that they could help tremendously in the event of secret exposure such as the Heartbleed attack, the fact, unfortunately, is that most systems do not deploy these cryptosystems.

Also, as seen in the previous section, many of the cryptographic tools we mention are forward-secure primitives. In terms of exposure-resilient cryptographic tools, we limit the scope of this paper to these primitives for brevity and simplicity. However, we point out here that forward security is not the only technique that can deal with exposure of secrets. Other tools include key-insulated cryptosystems [14], intrusion-resilient cryptosystems, threshold cryptosystems [11, 12], and proactive cryptosystems [15]. The first two groups require an additional computational device to achieve some weaker forms of security *before and after* the secret exposure, rather than only *before* the exposure as is the case with forward security. The last two groups require that multiple (usually more than two) parties work closely together when carrying out a task. These tools are also valid alternatives for dealing with Heart- bleed. They are, however, likely to be more complicated to set up in practice compared to forward-secure cryptosystems since they require additional coordination between a server and a helper device or between multiple servers.

5.1 Forward-secure key exchange

The helpful guanrantee with forward-secure key exchange is that an adversary who obtains the server's long-term secret key at time *t* still would not know any session key that was generated before time *t*.

The most popular KE protocol for SSL/TLS among server system administrators is the RSA-based KE protocol. In this protocol, the client chooses a value for the session key *K*, encrypts *K* with the server's public key *pk*, then sends the result to the server. Upon receiving this result, the server decrypts it using the long-term secret key *sk* to obtain *K*. Consequently, RSA-based KE protocol does not offer forward security: if the long-term secret key *sk* becomes exposed at some point, an adversary who has captured the data exchanged between the client and the server in the past can use *sk* to decrypt the appropriate segment of the protocol flows to obtain *K* and can in turn use *K* to decrypt the data encrypted with *K*.

In contrast, the KE protocol implemented for OpenSSL that is forward secure is the *Diffie-Hellman Ephemeral* (DHE) protocol [13]. Unfortunately, this protocol is not popular among system administrators. The most commonly cited reason is the claim that it is too computationally intensive. DHE is based on the basic Diffie-Hellman key exchange protocol shown in Fig. 1. The session key obtained at the end of the protocol is *K*.

The basic protocol shown in Fig. 1 does not keep adversary A from pretending to be the client or the server. For example, A can pretend to be a server "S" by simply engaging in the protocol exchanges with the client using "S" as its name. *Digital signatures* is used to address this problem: the server signs its protocol flow using its long-term secret key; the client then verifies, using the server's public key, that the protocol flow it receives has indeed been generated by the server.

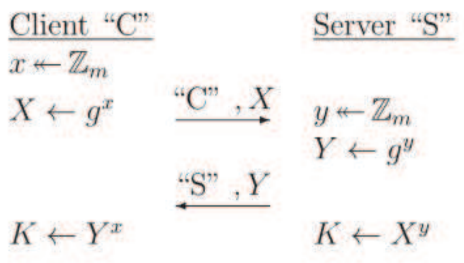


Fig. 1. The basic Diffie-Hellman key exchange. The cyclic group G generated by g is of order m. We make G, g, and m public. The notation s $\twoheadleftarrow$ S means the value assigned to s is chosen uniform randomly from the set S. An arrow $\leftarrow$ denotes an assignment.

5.2 Forward-secure digital signatures

As previously discussed, when a long-term secret key *sk* is exposed, the authenticity of all signatures purportedly generated using *sk* can no longer be trusted. This includes all certificates signed by the owner of *sk*. They must now be revoked. This problem can be addressed via forward-secure digital signatures.

In a forward-secure digital signature scheme [3], time is divided into periods, e.g. months, days, or weeks. The forward security guarantee is that, despite a long- term secret key exposure at time period *t*, signatures generated before *t* can still be trusted while those generated during and after t cannot be. The crux lies in the deterministic *key update* algorithm. Suppose that initially the server's secret and public keys are *sk* and *pk*, respectively. At each time period, *sk* gets updated while *pk* remains fixed. Since *pk* is unchanged, anyone can verify signatures using the single value of *pk* regardless of which time period the signature was generated. The secret key update algorithm is designed to be easily computed going forward (e.g. from time period *t* to *t* + 1) but difficult to go backward (e.g. from *t* + 1 to *t*). Consequently, if adversary *A* obtains sk_t, it can generate signatures on behalf of the server for time periods after *t* but not before (assuming that the secret keys for the time periods before *t* have been properly erased). Thus, certificates that have been issued (i.e. signed) before t remain trustworthy, and revocation can be avoided.

5.3 Forward-secure pseudorandom bit generator

Originally proposed by Bellare and Yee [5], a forward-secure pseudorandom bit generator (FS PRG) takes a state and returns the next state and the output. The way it is used is the following. The initial state is taken to be the original random seed. When a pseudorandom bits is needed, one invokes the PRG to update the seed and to obtain the pseudorandom bits to be used. The security guarantee is that, if adversary *A* obtains the current state s, it still would not be able to distinguish pseudorandom bitstrings that have been output prior to the exposure of the seed from outputs of a random function. This implies that, at the very least, *A* must not be able to obtain the states in the time periods prior to the exposure.

In a Heartbleed attack, if an adversary *A* obtains the current seed that is used by other cryptographic algorithms, for example to generate the session key, it can of course compromise the security of the current session. But if an FS PRG is used, *A* would not be able to derive the session keys for time periods prior to the break-in, thus leaving the security of these sessions intact.

5.4 Password-Based Authenticated Key Exchange

As discussed, since most clients do not have certificates, the most common authentication mechanism often involves the user sending his or her username and password to the server via SSL/TLS. In a Heartbleed attack, if adversary A obtains the username and password of a user from the server's memory, the user has no recourse other than to change the password.

If a *password-based authenticated key exchange* (PAKE) protocol is used, the user would not need to send the username and password over to the server and yet be able to establish a secret session key. In a PAKE protocol, the server and the user initially share a secret password *pw*, perhaps through a registration protocol or through out-of-band communication. Then, after the protocol completes, the user and the server end up with a secret session key. Many PAKE protocols have been proposed over the years. See for example [1, 4, 7–10].

5.5 Limited-use credit card numbers

A credit card number is a valuable piece of information. One can make purchases by giving a store a credit card number along with other accompanying information such as the appropriate expiration date and a "card security code" or a "card verification code," a 3-digit code found on the actual credit card itself. In an online purchase, all these pieces of information are sent to the server of the merchant. If the server is under a Heartbleed attack at the time the transaction occurs, an adversary can obtain this information from the server's memory.

Efforts have been made to limit the damage caused by credit card number exposure by limiting how long or how often a single credit card number can be used. These credit card numbers are sometimes called *disposable credit cards*. There are many approaches to implement this concept [16, 19, 24]. We discuss one simple example here. Rubin and Wright [16] propose that a user and his or her card issuer share a secret key *K* and use it to *authenticate and encrypt* the credit card number along with the rest of the authorizing information and restrictions regarding the current purchase. The resulting ciphertext *C* can be used in place of the credit card number during purchases.

We emphasize that in this case the information needs to be both authenticated and encrypted. Secrecy is necessary because the card number and other information must be kept secret. Authenticity is necessary because *C* is used as a token, and a token that can be minted by anyone, including in particular an adversary, is not very useful.

Furthermore, a scheme such as this needs a way to prevent a *replay* attack in which an adversary can simply capture the token *C* and simply reuse it to make other purchases. Including timestamps and other purchase-related information into the payload to be authenticated and encrypted can address this problem. Rubin and Wright further discuss this issue in [16].

We note here that, just like any other technical solution to real-world problems, there are usability issues associated with disposable credit cards. Consequently, more than a decade after their conception, they are still not prevalent.

6. Conclusion

Over the years, secret exposures have become frequent to a point where we should consider them to be common occurrences rather than rare events. Mechanisms that can mitigate secret exposures are of more importance now than ever. Cryptographers must forge ahead and come up with practical and efficient methods to deal with secret leakage. Practitioners must consider proposals that may at first glance seem esoteric and give security the high priority it deserves.

References

1. M. Abdalla and D. Pointcheval. Simple password- based encrypted key exchange protocols. In A. Menezes, editor, CT-RSA 2005, volume 3376 of LNCS, pages 191–208. Springer, Feb. 2005.
2. BBA, Inc. Testing for reverse heartbleed. http://blog.meldium.com/home/2014/4/10/ testing-for-reverse-heartbleed, Apr. 10, 2014.
3. M. Bellare and S. K. Miner. A forward-secure digital signature scheme. In M. J. Wiener, editor, CRYPTO'99, volume 1666 of LNCS, pages 431–448. Springer, Aug. 1999.
4. M. Bellare, D. Pointcheval, and P. Rogaway. Authenticated key exchange secure against dictionary attacks. In B. Preneel, editor, EUROCRYPT 2000, volume 1807 of LNCS, pages 139–155. Springer, May 2000.
5. M. Bellare and B. S. Yee. Forward-security in private-key cryptography. In M. Joye, editor, CT- RSA 2003, volume 2612 of LNCS, pages 1–18. Springer, Apr. 2003.
6. S. Bellovin. Problem areas for the ip security protocols. In Proceedings of the 6th USENIX Security Symposium 1996, pages 1–16, San Jose, CA, USA, July 22–25, 1996. USENIX Association.
7. S. M. Bellovin and M. Merritt. Encrypted key ex-change: Password-based protocols secure against dictionary attacks. In 1992 IEEE Symposium on Security and Privacy, pages 72–84. IEEE Com- puter Society Press, May 1992.
8. V. Boyko, P. D. MacKenzie, and S. Patel. Provably secure password-authenticated key exchange using Diffie-Hellman. In B. Preneel, editor, EURO- CRYPT 2000, volume 1807 of LNCS, pages 156–171. Springer, May 2000.
9. E. Bresson, O. Chevassut, and D. Pointcheval. Security proofs for an efficient password-based key exchange. In S. Jajodia, V. Atluri, and T. Jaeger, editors, ACM CCS 03, pages 241–250. ACM Press, Oct. 2003.
10. E. Bresson, O. Chevassut, and D. Pointcheval. New security results on encrypted key exchange. In F. Bao, R. Deng, and J. Zhou, editors, PKC 2004, volume 2947 of LNCS, pages 145–158. Springer, Mar. 2004.
11. A. De Santis, Y. Desmedt, Y. Frankel, and M. Yung. How to share a function securely. In 26th ACM STOC, pages 522–533. ACM Press, May1994.
12. Y. Desmedt and Y. Frankel. Threshold cryptosystems. In G. Brassard, editor, CRYPTO'89, volume 435 of LNCS, pages 307–315. Springer, Aug. 1990.
13. T. Dierks and C. Allen. RFC 2246 - The TLS Protocol Version 1.0. Internet Activities Board, Jan. 1999.
14. Y. Dodis, J. Katz, S. Xu, and M. Yung. Key-insulated public key cryptosystems. In L. R. Knud- sen, editor, EUROCRYPT 2002, volume 2332 of LNCS, pages 65–82. Springer, Apr. / May 2002.
15. R. Ostrovsky and M. Yung. How to withstand mobile virus attacks. In 10th ACM PODC, pages 51–59. ACM Press, Aug. 1991.
16. A. D. Rubin and R. N. Wright. Off-line generation of limited-use credit card numbers. In P. F. Syverson, editor, FC 2001, volume 2339 of LNCS, pages 196–209. Springer, Feb. 2001.
17. R. Seggelmann. Git commit diff. http://git.openssl.org/gitweb/?p=openssl.git;a=commitdiff;h=4817504, Jan. 1 2012.
18. R. Seggelmann, M. Tuexen, and M. G. Williams. Transport layer security (TLS) and datagram transport layer security (DTLS) heartbeat extension. http://tools.ietf.org/html/rfc6520, Feb. 2012.
19. A. Shamir. Secureclick: A Web payment system with disposable credit card numbers. In P. F. Syverson,

editor, FC 2001, volume 2339 of LNCS, pages 232–242. Springer, Feb. 2001.

20. 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.
21. N. Sullivan. The results of the cloudflare challenge. http://blog.cloudflare.com/ the-results-of-the-cloudflare-challenge, Apr. 11, 2014.
22. The OpenSSL Project. OpenSSL: Cryptography and SSL/TLS toolkit. http://www.openssl.org.
23. The OpenSSL Project. TLS heartbeat read overrun (CVE-2014-0160). http://www.openssl.org/news/secadv_20140407.txt, Apr. 07, 2014.
24. M. Trombly. American express offers disposable credit card numbers for online shopping. http://www.computerworld.com/s/article/49788/American_Express_offers_disposable_ credit_card_numbers_for_online_shopping, Sept. 7, 2000.