A CHALLENGING BUT FEASIBLE BLOCKWISE-ADAPTIVE CHOSEN-PLAINTEXT ATTACK ON SSL

Gregory V. Bard
University of Maryland, Department of Mathematics
College Park, MD, 20914, USA

Keywords: Blockwise Adaptive, Chosen Plaintext Attack (CPA), Secure Sockets Layer (SSL), Transport Layer Security (TLS), Cryptanalysis, HTTP-proxy, Initialization Vectors (IV), Cipher Block Chaining (CBC).

Abstract: This paper introduces a chosen-plaintext vulnerability in the Secure Sockets Layer (SSL) and Trasport Layer Security (TLS) protocols which enables recovery of low entropy strings such as can be guessed from a likely set of 2–1000 options. SSL and TLS are widely used for securing communication over the Internet. When utilizing block ciphers for encryption, the SSL and TLS standards mandate the use of the cipher block chaining (CBC) mode of encryption which requires an initialization vector (IV) in order to encrypt. Although the first IV used by SSL is a (pseudo)random string which is generated and shared during the initial handshake phase, subsequent IVs used by SSL are chosen in a deterministic, predictable pattern; in particular, the IV of a message is taken to be the final ciphertext block of the immediately-preceding message, and is therefore known to the adversary.

The one-channel nature of web proxies, anonymizers or Virtual Private Networks (VPNs), results in all Internet traffic from one machine traveling over the same SSL channel. We show this provides a feasible "point of entry" for this attack. Moreover, we show that the location of target data among block boundaries can have a profound impact on the number of guesses required to recover that data, especially in the low-entropy case.

The attack in this paper is an application of the blockwise-adaptive chosen-plaintext attack paradigm, and is the only feasible attack to use this paradigm with a reasonable probability of success. The attack will work for all versions of SSL, and TLS version 1.0. This vulnerability and others are closed in TLS 1.1 (which is still in draft status) and OpenSSL after 0.9.6d. It is hoped this paper will encourage the deprecation of SSL and speed the adoption of OpenSSL or TLS 1.1/1.2 when they are finially released.

1 INTRODUCTION

This paper outlines a vulnerability of all versions of SSL and also TLS 1.0 by means of a feasible blockwise-adaptive chosen-plaintext attack (BACPA). The attack proceeds as follows. After valuable low-entropy data has been transmitted by the target (e.g. a stock from a list of 2–1000 companies) the adversary inserts plaintext into the communications stream by inducing the target machine to transmit data designed to contain particular byte sequences. Based on the ciphertext values of these known plaintexts, the low-entropy data can be guessed. The probability of success depends on the type of data being targeted, with examples given below. Furthermore, it is shown that the position of this target data within block-boundaries *directly and significantly* impacts the number of guesses required to recover that datum.

The vulnerability described here was closed in TLS 1.1 and OpenSSL after 0.9.6d. Probably all other SSL implementations[1] are vulnerable. Since TLS 1.1 had only just exited draft status in late April 2006, most TLS deployments (at the time of the submission of this article) use version 1.0, which is vulnerable to this attack. Nonetheless, the main purpose of this paper is not to claim the existence of a major threat to computer security, but rather to disprove the myth that blockwise-adaptive chosen-plaintext attacks are totally infeasible and thus of theoretical interest only. Furthermore, we prove that it is easier to guess the values of low-entropy data when it is divided across block boundaries.[2] It is hoped this paper will highlight some of the disadvantages of Cipher Block Chaining, emphasize the importance of using

[1]There would be exceptions if others copy OpenSSL.
[2]We believe this has not previously been published.

V. Bard G. (2006).
A CHALLENGING BUT FEASIBLE BLOCKWISE-ADAPTIVE CHOSEN-PLAINTEXT ATTACK ON SSL.
In *Proceedings of the International Conference on Security and Cryptography*, pages 99-109
DOI: 10.5220/0002104100990109
Copyright © SciTePress

distinct keys for each source-destination pair, and illustrate the importance of using random numbers as initialization vectors.

The Attack Modern users are becoming aware that installing software of unknown origin is unsafe. For this reason, and many others, the concept of an "untrusted applet" running in a "sandbox" within the Java Virtual Machine was created (Gosling et al., 2005). These applets have highly restricted security privileges, but are commonly available on individual and commercial web-sites as utilities and games. If an attacker can manage to get a user to install a Java application (or application in any other language) the attacker can read the user's keystrokes and broadcast them—there is no need for cryptanalysis in this case. However, an untrusted applet does not have this privilege (Loeffler, 1997) (Gosling et al., 2005). It only receives notice ("events") from keystrokes pressed while the applet is on screen and "has the mouse focus." Therefore reading the user's keystrokes while he/she enters valuable data into other applications is impossible. However, one privilege of the untrusted applet is to open a TCP connection to the web server from whence it came (Loeffler, 1997). Imagine a user visiting a secure web-site for banking, and later playing a Java applet game. Naturally the socket for the secure transaction and the socket that the applet opens back to its server are unrelated. However, in the special case of a web-anonymizer, HTTP-proxy or virtual private network, all network traffic leaving the user's machine will travel over the same SSL socket[3]. Since they travel over the SSL link and in the same direction, the data in each case will be encrypted with the same secret key. This results in the targeted data, as well as data of the adversary's choosing (the chosen-plaintext), becoming available for use in verifying guesses of the targeted plaintext via Blockwise-Adaptive Chosen-Plaintext attack.

Blockwise-Adaptive Chosen-Plaintext BACPA was simultaneously discovered in 2001 by Bellare, Kohno, and Namprempre (Bellare et al., 2002) and by Joux, Martinet, and Valette (Joux et al., 2002). The BACPA differs from classical chosen plaintext attack (now termed "message-wise" or MCPA) in essentially one detail. In MCPA, the attacker can generate arbitrary messages for the target to encrypt, as part of a sequence of messages. In BACPA, the attacker can generate arbitrary blocks for the target to encrypt, inserted as part of an existing message (i.e., a sequence of blocks). However, four years after its discovery, BACPA has received very little notice outside the cryptographic community, despite several additional papers within it. Bellare, Kohno, and Namprempre (Bellare et al., 2002), in addition to outlining a theoretical model of potential adversary capabilities that is expanded upon in later papers (Fouque et al., 2004) (Boldyreva and Taesombut, 2004) (Fouque et al., 2003), loosely describe a general attack on the Secure Shell (SSH) first found by (Dai, 2002). That attack had a success probability[4] of $2^{-12.5}$ or 6.9×10^{-4}. This paper outlines an attack with a success probability that can approach 100%. We are aware of no other feasible attack under the BACPA model against an existing or proposed protocol, except for our previous paper (Bard, 2004).

The Versions of the SSL Protocol The Secure Socket Layer (SSL) (Freier et al., 1996) is currently one of the most widely-used methods for securing communication over the Internet[5]. There are two recent versions, SSL 2.0 (1995) and SSL 3.0 (1996). The successor protocol is Transport Layer Security (TLS), which includes version 1.0 (1999) (Dierks and Allen, 1999), and version 1.1 (Dierks and Rescorla, 2005) (2006). Also, a version 1.2 has been proposed (Dierks and Rescorla, 2006). Almost all secure web transactions (i.e. HTTPS) use either TLS 1.0 or SSL 3.0. For simplicity, we refer to all versions of SSL, as well as TLS 1.0, by saying "SSL." OpenSSL is an open source implementation of SSL. Versions of OpenSSL since 0.9.6d (Various,) have closed this vulnerability, as has TLS 1.1 (Dierks and Rescorla, 2005). All SSL deployments other than OpenSSL, including TLS 1.0, are believed to be currently vulnerable to this attack[6], though possibly other implementations have copied solutions used by OpenSSL. See Section 6.

Initialization Vectors Our attack relies on the fact that SSL currently mandates the use of a weak variant of the cipher block chaining (CBC) mode of encryption (Kaufman et al., 2002, Chap. 4). CBC mode requires a one-block initialization vector (IV) for each message that is encrypted. In "standard" cryptographic usage of CBC, a fresh, random IV is chosen for each message. In SSL, however, only the *initial* IV is chosen in a (pseudo)random manner; IVs for subsequent messages are simply taken to be the final block

[3]We have recently learned that an e-mail posted to the Certicom TLS mailing list in March of 2002 had also mentioned this possibility, though with fewer details (Rescorla, 2002).

[4]The success probability is stated to be equivalent to waiting for a collision on approximately 25 bits. See the last sentence of Section 3 of (Bellare et al., 2002).

[5]See (Kaufman et al., 2002, Chap. 19) for an excellent overview of SSL.

[6]Personal E-mails with E. Rescorla, co-author of the TLS 1.0, 1.1, and 1.2 RFCs

of the ciphertext corresponding to the immediately-preceding message. (This process is called “chaining the IVs.”) In particular, an attacker may know *in advance* the IV that is going to be used to encrypt the next message. We show that this enables an attacker mounting a chosen-plaintext attack to validate a guess as to the value of a particular plaintext block.

Consequences Since the adversary can validate a guess as to the value of a particular plaintext block, this attack violates the theoretical standard of Left-Or-Right Indistinguishability, whereby no polynomial time adversary may be able to distinguish between the encryption of two messages of his/her own choice, given the ciphertext, (with non-negligible advantage) (Goldwasser and Micali, 1984) (Bellare et al., 1997). On the more practical side, it possibly allows an attacker to determine a low-entropy string by repeatedly guessing all possible values for this string until the correct one is identified. Examples of such low-entropy information (2–1000 choices) include names of stocks, cities, users, or even PINs that have been previously encrypted (or, for example, knowing if a stock order is buy, sell, or stop-loss could be valuable information by itself). Given the use of SSL for transmitting exactly this sort of data, we believe this represents a potentially serious (but challenging) attack which should be addressed by any security group selecting an SSL implementation.

1.1 Related Work

Essentially this method has been used previously to attack SSH (Bellare et al., 2002) (Dai, 2002). In fact, the flaw attacked there is identical to the flaw attacked here (namely, setting IVs in a predictable way). Little discussion of feasibility or point-of-entry was given, and the probability of success was very low as already stated. On the other hand, that discovery gave birth to the blockwise-adaptive chosen-plaintext world.

Due to the similar structure of SSL and SSH, the related vulnerability in SSL was discovered soon after, independently by Moeller (Moeller,) and the author of this paper, in late 2002—8 months after the publications of the two original blockwise papers. Moeller’s work identifies the attack but does not show how it could be exploited. Moreover, our paper elaborates upon the attack by showing how low-entropy data, in particular, is easy to recover, and provides mechanisms by which to execute the attack. The author of this paper, in 2004, wrote of a similar BACPA attack against SSL but via a much more difficult and complex point-of-entry (Bard, 2004).

The changes to TLS between versions 1.0 and 1.1 were made partially in response to this class of vulnerability[7], as well as that of Vaudenay[8] (Dierks and Rescorla, 2005).

2 HIGH-LEVEL OUTLINE

We begin by briefly highlighting the minimal aspects of SSL needed to understand our attack at a high level. A more detailed treatment of the attack (and hence of SSL) is given in Section 3. A good survey of the SSL protocol is given in (Kaufman et al., 2002, Chap. 19).

The SSL protocol begins with a handshaking stage during which the parties agree on a protocol version, select cryptographic and (optionally) compression algorithms, perform optional authentication steps, and use public-key mechanisms to share secrets. The shared secrets, which include distinct symmetric keys and IVs for each direction of communication, can then be used for symmetric-key encryption and message authentication. While messages may optionally be compressed before encryption, few SSL implementations do so (Kaufman et al., 2002, Chap. 19), (Freier et al., 1996).

The SSL standard allows for symmetric-key encryption using either block ciphers or stream ciphers. Most implementations utilize block ciphers, and the vulnerability in this paper applies only when block ciphers are used. A block cipher is a keyed, invertible function (or *keyed permutation*) over strings of some fixed length called *blocks*; DES, for example, operates on 64-bit blocks. We write $F_{sk}(X)$ to represent the application of a block cipher using key sk to block X. To encrypt messages longer than one block in length, a *mode of encryption* must be used. SSL mandates the cipher block chaining (CBC) mode, which encrypts a message $P = P_1, \ldots, P_\ell$ (where the length of each P_i is the block-length of the cipher) as follows: given some IV denoted C_0, compute $C_1, \ldots, C_\ell$ sequentially via:

$$C_i = F_{sk}(P_i \oplus C_{i-1}).$$

In the general case of CBC, the resulting ciphertext is usually taken to be $C_0, \ldots, C_\ell$ although if the receiver already knows C_0 then it need not be transmitted. To decrypt, the receiver computes P_i for $i = 1$ to ℓ via:

$$P_i = F_{sk}^{-1}(C_i) \oplus C_{i-1}.$$

We note that it is considered “standard” security practice to choose a new, random IV for every message

[7]Attack denoted CBCATT in some TLS documentation (Moeller,), see also Section 6.2.3.2 of (Dierks and Allen, 1999) (Dierks and Rescorla, 2005).

[8]Another interesting attack on the use of CBC in SSL, discovered by Vaudenay, relates to the padding of messages, and is unrelated to the vulnerability in this paper (Vaudenay, 2001).

that is encrypted. However, the above definition of CBC does not force this to be the case. As we have mentioned already, SSL chooses all but the initial IV by setting it equal to the final ciphertext block of the preceding encrypted message; this is referred to as "chaining IVs across messages". (Thus, continuing the above example, the IV used for the next message would simply be C_ℓ.) SSL chooses the initial IV in a pseudorandom fashion which is not important for the purposes of the present attack.

The attack. Suppose an adversary who can mount a chosen-plaintext attack wants to verify a guess as to whether some plaintext block has a particular value. Specifically (continuing the above example), suppose an adversary who has observed the ciphertext $C_0, \ldots, C_\ell$ wants to determine whether plaintext block P_j is equal to P^*. Note that the adversary knows the IV (i.e., C_ℓ) that will be used when encrypting the next message. Consider now what happens if the adversary causes the sender to encrypt a message P' whose initial block P'_1 is equal to $C_{j-1} \oplus C_\ell \oplus P^*$. The first ciphertext block C'_1 is then:

$$\begin{aligned} C'_1 &= F_{sk}(P'_1 \oplus C_\ell) \\ &= F_{sk}((C_{j-1} \oplus C_\ell \oplus P^*) \oplus C_\ell) \\ &= F_{sk}(C_{j-1} \oplus (C_\ell \oplus C_\ell) \oplus P^*) \\ &= F_{sk}(C_{j-1} \oplus P^*). \end{aligned}$$

However, we also know that $C_j = F_{sk}(P_j \oplus C_{j-1})$. This implies that $C'_1 = C_j$ if and only if $P_j = P^*$, since F_{sk} is a permutation. In this way, an adversary can verify a guess P^* for the value of any plaintext block P_j. In particular, if the adversary knows that P_j is one of two possible values then the adversary can determine the actual value by executing the above attack a single time. Similarly, if the attacker knows that P_j is one of N possible values then by repeating the above attack $N/2$ times (on average) the adversary can determine the actual value of P_j. This already violates the standard notions of security for encryption (in terms of Left-Or-Right Indistinguishability, as mentioned in the Introduction (Goldwasser and Micali, 1984) (Bellare et al., 1997)). Moreover, this implies that an attack of this form can be used to determine the value of a low-entropy string in its entirety. (Note that, in practice, the block of plaintext containing the user's data also likely contains additional information such as headers, etc. However, it is also likely that this additional information is known to the attacker; for example, if the information is fixed padding, then an adversary can learn the format of this data from the web-page source code. We discuss this further in Section 4.)

Attack requirements. Focusing specifically on the case of an adversary trying to recover a user's targeted low-entropy data, we briefly highlight the requirements needed for the above-described attack to succeed; in Section 4 we discuss in more detail how these requirements are typically met in practice. First, the attacker must know which plaintext block j contains the desired information. All this means is that the adversary knows the format of the HTTPS transmission being targeted. Second, the adversary must know C_{j-1}. However, since the ciphertext travels over the Internet (in the clear!), this is not expected to be difficult. (In fact, if it is assumed difficult to obtain this information then there is little reason to use encryption in the first place.) Third, the adversary must know the value of the IV that is going to be used for the next message. However, we have noted already that because of the way SSL computes IVs, an attacker would actually obtain this information from the last ciphertext block of the previous message. Finally, the adversary must be able to insert a plaintext block of its choice into the first block of the next message to be transmitted. This is the most challenging part of the above attack.

The scenario. The mechanism by which the adversary executes the attack is summarized as follows. First, we assume that the user is connected to an HTTP-proxy, web anonymizing service, or virtual private network. This guarantees that all outbound Internet traffic will be routed over one SSL tunnel with a single secret key. Second, we assume the attacker can setup a site under his/her control, and create an applet for the user to open. In particular, applets have features, which will be described below in Section 4, that make them desirable for use. Third, a "reflector" or mechanism for observing the target's ciphertexts is set up. This is needed in all forms of cryptanalysis, so we presume it is available. Fourth, we assume that the user can be induced into opening the applet. This can be in the form of an email which persuades the user to go to the site, or perhaps an email that simply causes the applet to be opened upon viewing, via HTML encoded requests for the applet in the email body, using the <APPLET> tag. We fully acknowledge that these assumptions are non-trivial but they demonstrate that a chosen-plaintext attack is feasible.

Note that untrusted Java applets have very few privileges (Gosling et al., 2005), but one of those privileges is to open a TCP socket to the server from whence it came (Loeffler, 1997). They cannot, for example, simply read all of the user's keystrokes and export them to the adversary, as explained before. Since all outgoing traffic, in the case of a web anonymizer, HTTP-proxy, or virtual private network, travels over the same SSL link, this permits the applet to send and receive arbitrary data—which will become the chosen plaintexts of the attack.

3 ATTACKING SSL

Here, we simply note that there is nothing in the structure of SSL (such as extraneous headers or formatting information) which prevents the attack of the previous section from succeeding.

SSL sits between the Application and Transport layers, and so acts like a Session Layer in the OSI model. As such, SSL receives plaintext from the Application Layer as raw data. This plaintext is fragmented into blocks of length less than or equal to 2^{14} bytes. These blocks are optionally (but rarely) compressed[9] and are then processed and sent as follows:

- **Unencrypted Portion**:
  - `message type (8 bits);`
  - `major/minor version number (16 bits);`
  - `length counter (16 bits);`
- **Encrypted Portion**:
  - `plaintext fragment (`$\leq 2^{17}$` bits);`
  - `message authentication code (typically 160 bits);`
  - `padding (0--56 bits; ensures plaintext length is a multiple of block length);`
  - `padding length (8 bits);`

We stress that *the first block of the plaintext is the first block to be encrypted*. In particular, the header information that is prepended to the eventual transmission (i.e., the message type, major/minor version number, etc...). is *not* encrypted. Thus, as long as the adversary can set the first block of the plaintext fragment to some desired value (as discussed in the previous section), that block will be encrypted first and the attack will succeed.

We note that in SSH, some header data is prepended to the plaintext *before* encryption. This makes an attack such as the one outlined here more difficult in the context of SSH (Bellare et al., 2002) (Dai, 2002), since the adversary no longer has control over the first block of the plaintext that is eventually encrypted. Although it may be possible to work around these constraints (see (Bellare et al., 2002)), the attack is more difficult against SSH than it is against SSL (demonstrated by a success rate of $2^{-12.5}$ as compared to rates as high as 100% under certain circumstances).

4 FEASIBILITY

Several challenges must be surmounted before an adversary can successfully perform the attack that has been outlined here. The necessary steps to meet these challenges are listed below.

Using one common key. First and foremost, the data to be learned and the test cases must be encrypted with the same key. Furthermore, since the key for the block cipher is chosen at random each time an SSL socket is created, the data to be learned and the test cases must be transmitted through the same tunnel, and in the same direction. Luckily enough, when a web-tunneling service like an HTTP-proxy, or a VPN is used, all the out-going data travels over one SSL link, regardless of destination, and thus is encrypted with the same secret key.

Learning the plaintext format. Despite the length of a plaintext message, there are times when only a very small sequence of bytes is of critical importance. For example, a stock choice, a destination city for a geospatial inquiry, a user name, or data where only 2–1000 choices are considered likely, and known in advance (i.e., a “low-entropy string” in our terminology).

We have mentioned earlier that the adversary must somehow know which block of the plaintext contains the data of interest. Note, however, this is easily done by reading the source files for the pages that are used in sending the data. Discerning the format merely requires knowledge of HTTP, HTML, and CGI, and perhaps Javascript. Commonly available browsers have a “show page source” command, which displays the page’s HTML source code. Both the “form elements” which compile the user’s data, as well as the optional Javascript code which would verify its format, would thus be available to an attacker. While Javascript can alter HTML code, for example, to add hidden form elements, these would be visible in the Javascript code. The attacker need only read this code and the format is trivially derived.

Ensuring that the adversary’s chosen plaintext block is encrypted first. It is essential that the chosen plaintext, namely $P_i = P^* \oplus C_\ell \oplus C_{j-1}$, be the first encrypted block of the SSL datagram inside which it is found. However, this is easy, as we argue now.

Data is submitted to the SSL layer in the form of application-level messages, which are first aggregated into blocks of (at most) 2^{14} bytes in length. SSL does not respect message boundaries. If more than 2^{14} bytes are submitted, additional blocks are created; if several application level messages are submitted, they are concatenated in the buffer. However, in the absence of these two conditions, the data is encrypted and transmitted to the TCP layer immediately. Therefore, short intermittent messages would be encrypted and transmitted immediately. The structure of SSL packets guarantees, in the absence of concatenation, that the first block of the application message will be the first encrypted block of the SSL datagram.

[9] Our attacks do not apply when compression is used; however, we note that it is rarely used.

Concatenation only occurs when two messages arrive at the SSL layer simultaneously. However, since the packet source (the applet or game) is under the control of the adversary, the timing could be adjusted to guarantee that two packets never arrive during the required interval. Of course, packets could arrive from some other user activity or application. These timing failures can be avoided probabilistically by repeating each guess several times. Finally it should be noted that in the event concatenation does occur, if the game packet is the first packet, with other packets appended, no disadvantage is caused for the attack. Nonetheless, we acknowledge the timing difficulties.

Encrypting Chosen Plaintext. The attacker must force the encryption of particular data in any chosen plaintext attack, in this case based upon the previous ciphertext block, the guess of the targeted plaintext, and the ciphertext previous to the data to be guessed. Once this block is calculated, it must be inserted into the "plaintext stream."

Since our scenario of a user connecting via an HTTP-proxy or a VPN involves *all network traffic flowing to/from the target machine being encrypted by the same SSL connection*, with a distinct key for each direction, the attacker need only cause the chosen plaintext data to appear in an application-level datagram, traveling in the same direction as the targeted plaintext. However, there are other considerations.

- **The data must start the datagram.** Once the immediately previous ciphertext block is known, the very next plaintext block must contain the attacker's plaintext. There must be no blocks in between, otherwise our previous formulas fail. Therefore, it is essential that the test block (first block of new plaintext) be the first block of the datagram.
- **The data must be tolerated.** The data must not result in a crashed application, a dropped connection or error messages that would alert the user. Sending arbitrary bytes to an application via a datagram would almost surely violate that application protocol's datagram formatting rules. The data must "blend in" to the data that would have been otherwise sent.[10] For example, if the application has a header for each datagram, then its unlikely that the cryptanalysis blocks, being the first blocks of the datagram, will conform to that header's definition.
- **The application must anticipate many datagrams.** Since only one guess can be validated per packet, the application must anticipate a moderate number of packets (on the order of 1–500 for a guess among a set of 2–1000 choices).
- **Background Traffic** The attack will also be less likely detected if there is background traffic to/from the applet when the attack is not running.

[10]Special thanks to (a referee from a previous conference) for pointing out several challenges in this regard.

Given these four requirements, it seems desirable to write one's own application for the attack, rather than find an application that can tolerate the above. In particular, if the applet is programmed to search for a 128-bit "magic number" in each datagram, then it can discard all the data before and including that number. This way the cryptanalysis blocks can be safely ignored. The chances of the magic number occurring accidentally are almost infinitesimal.

Providing feedback to the Applet. Note that the adversary needs feedback from the reflector to the applet site (in particular, to inform it of the ciphertext blocks C_{j-1}, C_j, C_ℓ) in order to perform the attack. It is not expected to be difficult for an adversary's reflector to obtain these ciphertexts (after all, they are traveling on the Internet), but the information must somehow be communicated to the applet site.

There are two principal avenues through which this communication could occur. First, if the attack is being performed in a wireless environment, setting a wireless device to promiscuous mode is attractive. However, many systems use WEP (Wired Equivalent Privacy) though certainly not all (including, for example, the University of Maryland and American University campus networks, which are unencrypted).

Alternatively, if the adversary has access to the same Ethernet network on his/her own machine that the user is connected to, then the following can be executed. Perhaps a subordinate employee will use his/her private laptop with tcpdump and a special feedback script acting as a reflector, to capture his/her employer's SSL traffic. The game applet's server would only need to know where the feedback daemon is, receive the ciphertexts, and use it to compute new plaintexts. The search for the daemon could be through the use of a throw-away DNS address, or by hard-coding the IP address.

On the one hand this could be detected by an Intrusion Detection System as unauthorized traffic, but on the other, steganography could be used to embed the data in legitimate communications. Alternatively, since few Intrusion Detection Systems would be this sensitive, the reflector could simply pass the packets back in the clear.

Synchronization. The demands of synchronization are important since there can be no transmitted data between the "last ciphertext block" and the encrypted attacking plaintext. There may be many ways to achieve this, but the game/applet server could signal that it is ready by sending a special packet. The reflector responds with the most recently transmitted ciphertext, which hopefully travels with minimal delay to the applet. Upon receipt of this, the applet transmits a packet, with or without game data. The first 8 or 16 bytes of the packets (one encrypted word) is not game data, but the chosen-plaintext. If no data (from any application) was transmitted on the tunnel in this time, then this stage of the attack succeeds and one guess can be checked. If data was transmitted on the tunnel in this time, then the plaintext loses its meaning in this context, but no harm is done and the attack can be attempted again. Since the Poisson Distribution is known to be a good model of user packet generation during web-browsing and other non-streaming Internet tasks, large gaps between packets will be expected. This is especially true if the game is interesting and the user is not surfing other sites while playing it.

If the user will play for as little as 15 minutes, many packets can make the round trip in this time. Since the "go signal", the reflector's "last seen ciphertext", and the "guess" must all be exchanged, naturally with acknowledgments (since we are using TCP), six round-trip times are required. Rarely is a round-trip more than 250 ms, so each "guess" takes 1.5 seconds, or 40 guesses can be made per minute. This comes to 600 guesses per short 15 minute game. Some of these guesses will be clobbered by traffic from other sockets, but others should succeed. Even if only one-third of the packets get through (which is very pessimistic), a choice from a list of 200 options will be found with absolute certainty, and a choice from a list of 10,000 options (e.g. a four digit PIN number) will be found with a probability of 2%.

Summary The requirements listed in this section are non-trivial. The timing and other network packet structure considerations will result in some guesses failing to serve their purpose. But we believe that we have demonstrated the potential feasibility of this sort of attack. This further demonstrates that BACPA is of more than purely theoretical interest.

5 RECOVERING PIN'S

Here we show how even a moderately small entropy string ($\leq$10,000 choices) can be easy to recover due to segmentation that occurs when the target data falls on a block boundary. (We assume throughout this section that the data surrounding the target information is known; see above). Since it is now demonstrated that the adversary has the ability to verify guesses of plaintext blocks, one can imagine that an adversary can attempt to guess the value of a valuable target low-entropy string either by exhaustive search (in the case of Personal Identification Numbers or PINs) or by more efficient context-specific schemes (using dictionary based attacks versus passwords, for example).

If we assume for simplicity that the data is chosen uniformly from a space of size S, then the expected number of guesses needed before determining the string is $S/2$. (Note that in the case of passwords chosen by a user, the entropy is likely to be much lower than would be indicated by the length of the password alone. In particular, an 8-character password model typically has entropy much lower than 64 bits). For example, a 4-digit PIN can be determined with (on average) 5,000 guesses. If only 100 guesses can be made, the probability of success is 1%, which is low, but certainly represents a feasible attack—we shall see momentarily that this can be improved greatly.

Block Cipher Choices: Recall that AES was not in existence when SSL 3.0 was originally specified (1996), nor was it finalized when TLS 1.0 was released (1999). Both SSL 3.0 and TLS 1.0, the standardized versions now available, do not include AES in their specification (Freier et al., 1996) (Dierks and Allen, 1999). Likewise, TLS 1.1 does not have AES as a built-in option, but it is available via some extension documents (e.g., (Modadugu and Rescorla, 2006)). Finally TLS 1.2 will have AES built in, as this was one of two principle changes between 1.1 and 1.2, the other relating to the choice of hash functions (Dierks and Rescorla, 2006). Therefore, essentially all SSL/TLS transactions at this time use DES or 3-DES, both with a 64-bit plaintext and ciphertext block length (though with an effective key size of 56 or 112 bits).

Split Target Data: However, assume 3-DES is being used as the block-cipher in the SSL transaction. Then the plaintext blocks are 64 bits or 8 bytes. There is a 12.5% probability that the four bytes of pin-data will be divided exactly in the middle of a block boundary.[11] *Since each block can be guessed independently*, 50 guesses are required for the 100 options on the left, and 50 guesses are required for the 100 options on the right. This use of 100 guesses results in a 25% chance of success, since both halves must be correct—rather than a mere 2% above. Alternatively, fewer guesses can be made if a lower probability of success is tolerated. Even a 3-1 or 1-3 split results in a major distortion of the number of required guesses. And there is a 37.5% chance that such a split (1-3, 2-2 or 3-1) will occur. See the Appendix for a discussion of applying this technique to passwords.

[11] One must assume that enough traffic has gone by that the "indentation" or "offset" within the stream of the valuable data is unpredictable, and therefore can be treated as a uniformly distributed random variable.

One should be careful to distinguish forms where passwords are entered all at once, and then submitted via SSL, from the normal SSH method where passwords are entered one keystroke at a time. In the latter case, this trick of breaking up the password would not work—a single byte is never broken up [12].

6 SOLUTIONS

It has been noted that TLS 1.1 and OpenSSL (after 0.9.6d) are not vulnerable to this attack, for reasons detailed below.

TLS 1.1 and Explicit IVs The TLS 1.1 protocol (Dierks and Rescorla, 2005) fixes this vulnerability by using explicit IVs. That is to say, each message has one more ciphertext block than plaintext blocks. This first ciphertext block is the IV, determined as a (pseudo)random number. As we have stated several times already, this is the accepted way to encrypt using CBC. Since the attacker does not know this value in advance, this attack cannot be executed. Using the formula for CBC, it is easy to see that having an Explicit IV is identical to adding an additional plaintext block of all zeroes to the start of each message, which the receiver knows to discard, and chaining the IVs from message to message (Moeller,).

If generating truly random bits is a concern (say, for reasons of efficiency), it is easy to generate a pseudorandom IV in any of a number of ways. For example, instead of simply using C_ℓ (i.e., the last block of the preceding ciphertext) as the IV, the protocol could use $H(C_\ell|sk)$ where sk is the shared secret key used for encryption and H is a cryptographic hash function.

Single Block Nonces This solution is mentioned because it can be used in other applications that use CBC, to protect them from BACPA, and helps to explain the solution used by the OpenSSL community. As was noted previously, the adversary's guess must be the first block of each message. Introducing a one block nonce—which would always be discarded before reading the message—would make this impossible. The nonce does not even need to be random. Suppose the nonce was always the all-zero string, and the previous message ended on block C_i. Then the adversary will submit a block P_{i+2} based on a guess G for block g. This will be output in block $i+2$, where $i+1$ is the nonce. Therefore the submitted plaintext, from the attack formula given previously would be

$$P_{i+2} = C_{i+1} \oplus C_{g-1} \oplus G$$

But C_{i+1} has not been transmitted yet. It equals

$$C_{i+1} = F_{sk}(C_i \oplus 0000 \cdots 0)$$

Therefore, C_{i+1} can only be determined if $F_{sk}(C_i)$ has been calculated before, or the adversary guesses the $F_{sk}(C_i)$. Since $F_{sk}(\cdot)$ is a function family believed to be pseudorandom, this guess will be correct with negligible probability. Likewise, since the space of all possible ciphertexts is 2^{64} or 2^{128}, the probability of a repetition is low. As mentioned earlier, in the case of the all zero plaintext, this is identical to explicit IVs. The TLS specification for version 1.1 allows for this solution, in addition to Explicit IVs, but recommends that the nonce be pseudorandom, generated for each message independently (Dierks and Rescorla, 2005).

OpenSSL and the Empty Message A slight variation of the above is used by OpenSSL after version 0.9.6d (Various,). An empty message, which consists of no plaintext, but only padding and a hash, is prepended to each set of messages before encryption. The "extra parts", namely the padding and hash, are encrypted, and so form the throw-away blocks similar to the nonce above. Since the adversary's chosen blocks are no longer the first to be encrypted, this attack becomes impossible. What is interesting about this solution is that it does not require any major changes at all to the SSL standard, and fulfills the present definition of SSL as written. The only point to mention is that some SSL clients will declare an error if an empty message is received. This error message need only be suppressed, which is a minor change.

Compression Note that an immediate way to prevent the attack suggested here is to turn compression on (as we have noted, an attack of the sort suggested here is much more difficult — if not impossible — if compression is used). However, this requires that peers only communicate with others who also use compression (or else an adversary connecting to the honest party could mount a "chosen-protocol attack" in which they claim to be unable to use compression). On the other hand, blocking users not configured for compression would limit inter-operability with deployed versions of SSL.

7 CONCLUSIONS

The attack presented here is not so easy that it can be done on the spur of the moment by the typical

[12] Personal E-mails with E. Rescorla, co-author of the TLS 1.0, 1.1, and 1.2 RFCs (Dierks and Allen, 1999) (Dierks and Rescorla, 2005) (Dierks and Rescorla, 2006)

hacker. However, while the attack is challenging to carry out, the success probability and relatively low numbers of datagrams required should be sufficient to motivate the SSL community to migrate away from TLS 1.0 and SSL, to OpenSSL after 0.9.6d, or TLS 1.1/1.2 when they are finally released. Moreover, by demonstrating the existence of this attack on a real-world protocol, which corresponds to the theoretical definition of blockwise-adaptive chosen-plaintext attack, we prove that the BACPA model is not sterile, but is useful for modeling adversarial capabilities.

Moreover it is hoped that this work will take a step toward opening the dialog between protocol designers and theoretical cryptographers, and stimulate discussion between these two camps which are otherwise independent. Finally, there are other uses of CBC similar to that of SSL, and this attack shows that those applications should also use explicit IVs or another solution listed here (e.g. Datagram Transport Layer Security or DTLS (Modadugu and Rescorla, 2004)).

ACKNOWLEDGEMENTS

Thanks to Prof. Jonathan Katz for suggesting the problem, for helpful discussions, and for substantial help editing this document. Thanks also to Ruggero Morselli, Patrick Studdard, Radostina Koleva, Zhongchao Yu, Susan Schmoyer and Prof. Lawrence Washington (all of the University of Maryland), for reading and commenting extensively on early versions of this paper. Valuable feedback was received from Daniel Brown of CertiCom, Eran Tromer of the Weizmann Institute, Prof. Bodo Moeller of UC Berkeley, and Jack Lloyd of randombit.net, about an earlier paper outlining a different attack on this same SSL vulnerability (Bard, 2004). Most importantly we would like to thank Eric Rescorla, co-author of the TLS RFCs (Dierks and Allen, 1999) (Dierks and Rescorla, 2005), (Dierks and Rescorla, 2006), for his extended correspondences via email that helped this work immensely.

REFERENCES

Bard, G. (2004). The vulnerability of ssl to chosen-plaintext attack. Cryptology ePrint Archive, Report 2004/111. `http://eprint.iacr.org/`.

Bellare, M., Boldyreva, A., Knudsen, L., and Namprempre, C. (2001). On-line ciphers and the hash-cbc construction. In *Lecture Notes in Computer Science*. Advances in Cryptology— CRYPTO'01, Springer-Verlag.

Bellare, M., Desai, A., Jokipii, E., and Rogaway, P. (1997). A concrete security treatment of symmetric encryption: Analysis of the des modes of operation. In *Symposium on the Foundations of Computer Science (FOCS'97)*. IEEE.

Bellare, M., Kohno, T., and Namprempre, C. (2002). Provably fixing the ssh binary packet protocol. In *Conference on Computer and Communications Security (CCS'02)*. ACM.

Bellare, M. and Namprempre, C. (2000). Authenticated encryption: Relations among notions and analysis of the generic composition paradigm. In *Lecture Notes in Computer Science*. Advances in Cryptology— ASIACRYPT'00, Springer-Verlag.

Boldyreva, A. and Taesombut, N. (2004). On-line encryption schemes: New security notions and constructions. In *Cryptographer's Track*. RSA Conference.

Dai, W. (2002). An attack against ssh2 protocol. Email to the ietf-ssh@netbsd.org email list.

Dierks, T. and Allen, C. (1999). The tls protocol, version 1.0. Technical Report RFC 2246, Internet Engineering Task Force.

Dierks, T. and Rescorla, E. (2005). The tls protocol, version 1.1. Technical Report RFC 2246-bis-11, Internet Engineering Task Force.

Dierks, T. and Rescorla, E. (2006). The tls protocol, version 1.2. Technical Report RFC 4346-bis-00, Internet Engineering Task Force.

Dworkin, M. (2001). Recommendation for block cipher modes of operation: Methods and techniques. Technical Report NIST Special Publication 800-38A, National Institute of Science and Technology.

Dworkin, M. (2002). Recommendation for block cipher modes of operation: The rmac authentication mode, methods and techniques. Technical Report NIST Special Publication 800-38B, National Institute of Science and Technology.

Fouque, P., Joux, A., and Poupard, G. (2004). Blockwise adversarial model for on-line ciphers and symmetric encryption schemes. In *Lecture Notes in Computer Science*. Advances in Cryptology— SAC'04, Springer-Verlag.

Fouque, P., Martinet, G., and Poupard, G. (2003). Practical symmetric on-line encryption. In *Lecture Notes in Computer Science*. Advances in Cryptology— FSE'03, Springer-Verlag.

Freier, A., Karlton, P., and Kocher, P. (1996). The ssl protocol, version 3.0. Technical report, Transport Layer Security Working Group Internet Draft.

Gligor, V. and Donescu, P. (2001). Fast encryption and authentication: Xcbc encryption and xecb authentication modes. In *2nd NIST Workshop on AES Modes of Operation*. National Institute of Science and Technology.

Goldwasser, S. and Micali, S. (1984). Probabilistic encryption. *Journal of Computer and System Sciences*.

Gosling, J., Joy, B., Steele, G., and Bracha, G. (2005). *The Java(TM) Language Specification*. Addison-Wesley Professional, third edition.

Joux, A., Martinet, G., and Valette, F. (2002). Blockwise-adaptive attackers: Revisiting the (in)security of some provably secure encryption models: Cbc, gem, iacbc. In *Lecture Notes in Computer Science*. Advances in Cryptology— CRYPTO'02, Springer-Verlag.

Kaufman, C., Perlman, R., and Speciner, M. (2002). *Network Security: Private Communication in a Public World*. Prentice Hall, second edition.

Knudsen, L. (2000). Block chaining modes of operation. In *Symmetric Key Block Cipher Modes of Operation Workshop*. National Institute of Science and Technology.

Krawczyk, H. (2001). The order of encryption and authentication for protecting communications (or: How secure is ssl?). In *Lecture Notes in Computer Science*. Advances in Cryptology— CRYPTO'01, Springer-Verlag.

Lipmaa, H., Rogaway, P., and Wagner, D. (2000). Comments to nist concerning aes modes of operation: Ctr-mode encryption. In *Symmetric Key Block Cipher Modes of Operation Workshop*. National Institute of Science and Technology.

Loeffler, S. (1997). Using flows for analysis and measurement of internet traffic. Master's thesis, Institute of Communication Networks and Computer Engineering of the University of Stuttgart. `http://www.mathematik.uni-stuttgart.de/~floeff/diplom/report/node62.html`.

Modadugu, N. and Rescorla, E. (2004). The design and implementation of datagram tls. In *Network Distributed System Security Conference*.

Modadugu, N. and Rescorla, E. (2006). Aes counter mode cipher suites for tls and dtls. Technical report, Internet Engineering Task Force.

Moeller, B. Security of cbc ciphersuites in ssl (tls problems and counter-measures. Posting on the Open SSL Project's website. `http://www.openssl.org/~bodo/tls-cbc.txt`.

Rescorla, E. (2002). [ietf-tls] re: Rfc 2246-bis open issues. Email to the ietf-tls@lists.certicom.com email list. `http://www.imc.org/ietf-tls/mail-archive/msg03341.html`.

Various. Various documents at the Open SSL web-site. `http://www.openssl.org/`.

Vaudenay, S. (2001). Security flaw induced by cbc padding applications to ssl, ipsec, wtls, ... In *Lecture Notes in Computer Science*. Advances in Cryptology— EUROCRYPT'02, Springer-Verlag.

A GENERAL PROBLEMS WITH CBC

The Cipher Block Chaining (CBC) mode of encryption was first proposed as a mode for DES, the Data Encryption Standard (Bellare et al., 1997). However, in the three decades that have passed since that time, much research has taken place in both adversarial modeling and modes of encryption (Bellare et al., 2001) (Dworkin, 2001) (Dworkin, 2002) (Fouque et al., 2003) (Gligor and Donescu, 2001) (Knudsen, 2000) (Lipmaa et al., 2000).

A mode of encryption is an algorithm for defining how the block cipher will be used to produce ciphertexts when the plaintext is of length longer than one block. For example, CBC has the formula $C_i = F_{sk}(C_{i-1} \oplus P_i)$ where C_0 is an initialization vector, and Counter Mode (CTR) has the formula $C_i = F_{sk}(i + i_0) \oplus P_i$, where i_0 is an initialization vector (Knudsen, 2000) (Lipmaa et al., 2000).

We note the following disadvantages of CBC.

- CBC is vulnerable to blockwise-adaptive chosen-plaintext attack, while CTR is not.
- An error in one block of CBC renders unreadable the remainder of the message, while an error in one block of CTR only renders that block unreadable.
- CBC cannot be parallelized, as can CTR. In CBC each block depends on the encryption of the block before it. In CTR, each block is encrypted independently.
- CBC is subject to the padding attack of Vaudenay (Vaudenay, 2001), but this is avoidable if one pads according to the algorithm given in that paper. In CTR, one can pad arbitrarily and have the padding length as an extra plaintext block at the end.
- CBC offers no protection versus Chosen Ciphertext Attack (CCA), as would HPCBC, XCBC, or OCB (Bellare et al., 2001) (Gligor and Donescu, 2001). However, Counter Mode also offers no CCA protection.
- An initialization vector need only be calculated once in CTR mode, not per message as in CBC. The counter i can be statefully maintained, and so will not repeat until 2^{64} or 2^{128} blocks.
- While this does not necessarily apply to SSL, the $F_{sk}(i + i_0)$ can be pre-computed, leaving only XOR operations with the plaintext to compute the ciphertext. If data needs to be transmitted only occasionally, but urgently when ready, this can be an advantage.

Since SSL uses Message Authentication Code (MAC) algorithms for the datagrams, the malleability attacks and other CCA attacks that plague CTR mode are of no interest. Therefore it is clear that CTR and not CBC would be a better choice for future versions of TLS and SSL, as CTR has advantages and no disadvantages over CBC (Lipmaa et al., 2000).

Other Problems with SSL's Encryption: We note that the mode of encryption should probably be changed to address other concerns as well. For example, it is well-known that applying a message authentication code to the ciphertext itself *after* encryption is preferable to applying it to the message *before* encryption (Bellare and Namprempre, 2000) (Krawczyk, 2001). Currently, SSL does the latter rather than the former.

B THE APPLICATION OF SPLITTING BLOCKS TO GUESSING PASSWORDS

While the attack presented in this paper really can only provide for up to about 1000 guesses at best (a long game with little background traffic), under ideal conditions, other chosen plaintext attacks might allow for several more. Therefore it is interesting to point out the effect of splitting upon passwords. If the printable ASCII character set of 95 choices is used, and the passwords are 8 bytes long, and nearly randomly chosen (very generous assumptions), then there are $95^8 = 6.6 \times 10^{15} = 2^{52.6}$ possible passwords, and $2^{51.6} \approx 3.4 \times 10^{15}$ guesses would be required in expectation.

However, the probability of the password not being broken in two is 12.5%. Even with AES and 16-byte blocks, $\frac{7}{16} = 43.8$% will be broken into pieces. If divided down the center, (regardless of 64-bit or 128-bit blocks) the number of guesses expected would be

$$\frac{2^{\frac{52.6}{2}} + 2^{\frac{52.6}{2}}}{2} = 2^{26.3} = 8.26 \times 10^7$$

Since 3.4×10^{15} guesses are expected in the unsplit case, and 8.3×10^7 in the split case, the attack becomes 4.1×10^7 times faster when the password is split. While our present attack scenario is not suited for anywhere near this number of guesses, there may be similar scenarios which can tolerate a few tens of thousands guesses, and if a few thousand users are targeted, then at least one password recovery would be expected, if not more.