Vulnerability of SSL to Chosen-Plaintext Attack

Gregory V. Bard*

May 11, 2004

Abstract

The Secure Sockets Layer (SSL) protocol is widely used for securing communication over the Internet. When utilizing block ciphers for encryption, the SSL standard mandates the use of the cipher block chaining (CBC) mode of encryption which requires an initialization vector (IV) in order to encrypt. Although the initial 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. We show that this introduces a vulnerability in SSL which (potentially) enables easy recovery of low-entropy strings such as passwords or PINs that have been encrypted. Moreover, we argue that the open nature of web browsers provides a feasible "point of entry" for this attack via a corrupted plug-in; thus, implementing the attack is likely to be much easier than, say, installing a Trojan Horse for "keyboard sniffing". Finally, we suggest a number of modifications to the SSL standard which will prevent this attack.

Keywords: Chosen Plaintext Attack, SSL/TLS, Cryptanalysis.

1 Introduction

The Secure Sockets Layer (SSL) protocol[1] is currently one of the most widely-used methods for securing communication over the Internet (see [8, Chap. 19] for an excellent overview of SSL). Here, we demonstrate a weakness in SSL which potentially allows an attacker mounting a chosen-plaintext attack to gather information about the plaintext being encrypted. In particular, the attack potentially enables an adversary to easily recover low-entropy information such as passwords or PINs that have previously been encrypted. Given the use of SSL for transmitting exactly this sort of data, we believe this represents a potentially serious attack which should be addressed in future versions of SSL.

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 [8, 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 of the ciphertext corresponding to the immediately-preceding message. In particular, an attacker may know *in advance* the IV that is going to be used to encrypt the next message. We show that

*`gregory.bard@ieee.org`. Dept. of Computer Science, University of Maryland.

[1]The attack outlined here applies to both the most recent versions of SSL (i.e., version 3.0) [5] and TLS [4]. For convenience, we simply refer to these generically as "SSL".

this enables an attacker mounting a chosen-plaintext attack to validate a guess as to the value of a particular plaintext block. Besides the fact that this already violates the definition of secure encryption [6, 1], it also allows an attacker to completely determine the value of a low-entropy string such as a password or a PIN (we call such short but valuable data a "nugget") by repeatedly guessing all possible values for this string until the correct one is identified.

While an attack of the type we outline here has been suggested previously in other contexts (see the discussion of previous work below), our work is significant insofar as it impacts the security of a widely-used product and should be taken into account in future revisions of the SSL standard. Furthermore, we believe our discussion of the means for executing the attack on SSL (in particular, our suggestion that the open nature of web-browser plug-ins could easily provide a "point of entry" for chosen-plaintext attacks) is of independent interest. Finally, the fact that the vulnerability is present in SSL even following recent work demonstrating similar attacks only further indicates the need to publicize these attacks within the security community.

This paper is structured as follows. Following a discussion of related work, we provide a "high-level," cryptographic description of our attack in Section 2. Section 3 focuses on "low-level" details of the attack, and shows that the attack as outlined can actually be implemented against SSL/TLS. As mentioned earlier, the attack allows an adversary to confirm guesses as to the value of a plaintext block at a rate of one guess per block of chosen plaintext. Thus, low-entropy nuggets are particularly vulnerable. We show in Section 5 how our attack can be used to recover even moderately-long nuggets when these nuggets are "split" across block boundaries (which is expected to occur frequently in practice). In Section 4 we discuss the feasibility of our attack when SSL is used within a web browser (as is almost always the case). Although the vulnerability we expose here is not trivial to exploit, we believe the reasons enumerated in Section 4 indicate that the attack does represent a potential concern. In particular, Section 4 highlights why this attack — which requires the adversary to convince a user to install an "infected" browser plug-in — is likely to be easier than an attack which requires the adversary to convince a user to install Trojan Horse software at the operating system level (say, to "sniff" the user's password via keystroke capture). We conclude the paper in Section 6 with some recommendations for patching the vulnerability exploited here.

1.1 Related Work

At its core, the attack described here is an example of what has recently been termed a "blockwise-adaptive" attack [7]. In contrast to a "standard" chosen-plaintext attack where messages are viewed as atomic, in a blockwise-adaptive attack an adversary is assumed to have the additional ability to insert plaintext blocks within some longer message *as that message is being encrypted* (we refer to [7] for a discussion). Although our attack does not exactly follow this paradigm, one may cast our attack in this light due to the fact that the IV for each message is taken to be the final block of the ciphertext corresponding to the preceding message. In particular, this is precisely how CBC would operate were it to encrypt consecutive messages as one, longer message; in other words, this "feature" of SSL is exactly what makes a blockwise-adaptive attack feasible.

Essentially this sort of attack has been used previously to attack SSH [2]. In fact, the flaw attacked there is identical to the flaw attacked here (namely, setting IVs in a predictable way). Still, we believe we provide more convincing evidence as to why the attack is actually plausible in the context of SSL, particularly when low-entropy data (such as PINs and passwords) are encrypted. Furthermore, as we have mentioned earlier, the fact that this vulnerability remains in deployed software only serves as evidence that attacks of this sort still need to be publicized within the security/cryptography communities.

2 High-Level Outline of The Attack

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 [8, 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 symmetric keys and IVs for each direction of communication, can then be used for symmetric-key encryption and message authentication. We note that while messages may optionally be compressed before encryption, few SSL implementations currently do so [8, Chap. 19], [5].

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 permutation over strings of some fixed length called *blocks*; DES, for example, operates on 64-bit blocks. We represent application of a block cipher using key sk to block X by writing $F_{sk}(X)$. 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:

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

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 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 some 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 a particular plaintext block has a particular value. Specifically (continuing the above example), say an adversary who has observed the ciphertext $C_0, \ldots, C_ell$ wants to determine whether plaintext block P_j is equal to some string 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 computed as:

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

However, we also know that $C_j = F_{sk}(P_j \oplus C_{j-1})$. This implies that $C'_1 = C_j$ iff $P_j = P^*$. 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. Besides already violating the standard notions of security for encryption [6, 1], this implies that an attack of this form can be used to determine the value of a nugget (say, a short password) in its entirety. (Note that, in practice, the block of plaintext containing the user's password 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 than 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 password or PIN, 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, however, 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. We believe, however, that such an attack is possible if the adversary can convince an unwitting user to use a plug-in of the adversary's design. We comment in Section 4 why this is expected to be significantly easier than convincing a user to install other malicious software, such as a keyboard sniffer, directly.

We note that a malicious browser plug-in *cannot* be used to directly "sniff" the user's password; thus, the above vulnerability does indeed represent a new "avenue of attack" for an adversary. Figure 1 shows the data-flow in a typical web-page with forms and SSL. A web-page contains data of different types, some of which (being pure HTML) would be displayed directly by the browser. Other data comes from files that the browser is programmed to handle, such as image files. Many other file formats exist, however, and the browser uses plug-ins to process them. In the case of common formats like sound files, the plug-in is probably already installed. In the case of a rare (or, for the attack, newly-invented) file format, the plug-in would have to be downloaded. Each plug-in interacts with the browser and its own file type (paths A and B in Figure 1). The keystrokes of the user on the other hand, after passing through the operating system, pass directly to "form elements" (path E) and then to the browser. In particular, a malicious plug-in does *not* have direct access to these keystrokes.

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

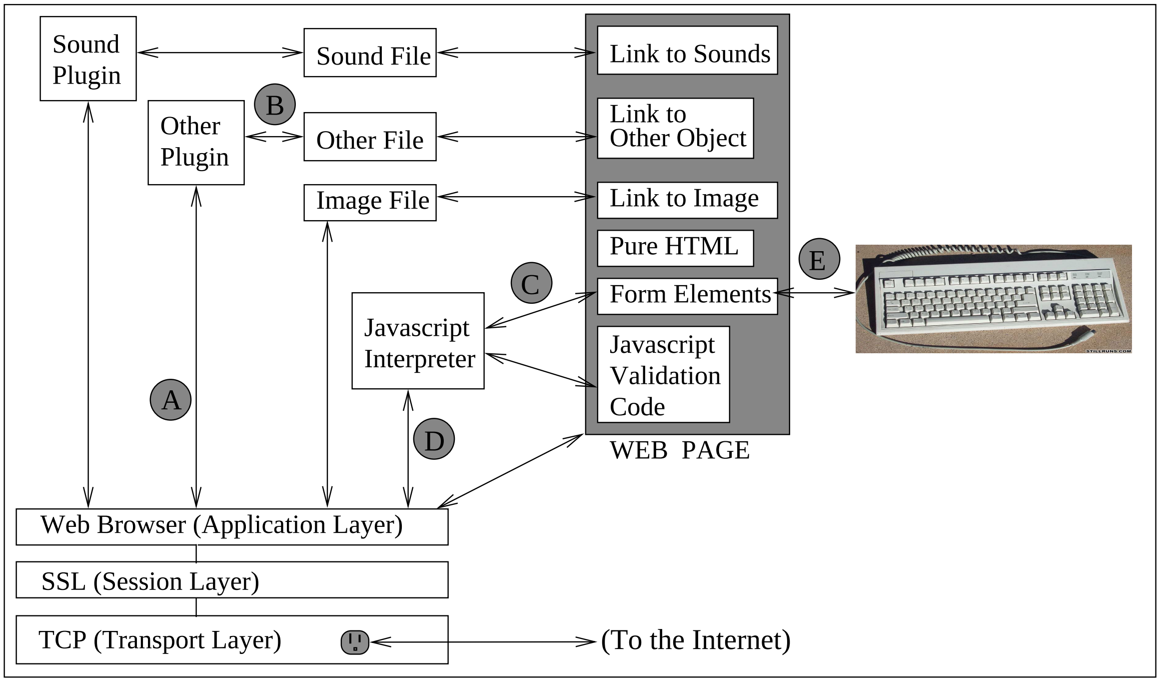

Figure 1: The data-flow of a web-page with an SSL-enabled form.

rarely) compressed[2] and are then processed and sent as follows:

- **Unencrypted Portion**:
  - message type (one byte);
  - major/minor version number (two bytes);
  - length counter (two bytes);
- **Encrypted Portion**:
  - plaintext fragment (arbitrary length $\leq 2^{14}$ bytes);
  - message authentication code (typically 20 bytes);
  - padding (0 to 7 bytes; ensures that the plaintext length is a multiple of the block length)
  - padding length (one byte)

We stress that *the first block of the plaintext is indeed 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 pre-pended to the plaintext *before* encryption. This makes an attack such as the one outlined here more difficult in the context of SSH [2], since the

[2] The attacks in this paper do not apply directly when compression is used; however, we have already noted that compression is rarely used. In any case, compression is unlikely to be used if only a short password/PIN is sent.

adversary no longer has complete control over the first block of the plaintext that is eventually encrypted. Although it may be possible to work around these constraints (see [2]), the attack is much more difficult against SSH than it is against SSL.

4 Feasibility of Implementing the Attack

Several challenges must be surmounted before an adversary can successfully perform the attack that has been outlined here. However, the nature of web browsers makes most of these challenges easy to deal with. The necessary steps to circumvent these challenges are listed below.

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 superuser password, a personal identification number or a dollar amount (i.e., a nugget in our terminology). We have mentioned earlier that the adversary must somehow know which block of the plaintext contains the nugget of interest. Note, however, this is easily done by reading the source files for the pages that are used in sending the nugget. Discerning the format merely requires knowledge of the HTTP, HTML, and CGI protocols, 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. The attacker need only read this and the format is trivially derived.

Using plug-ins to perform chosen-plaintext attacks. The next problem would be to arrange for the chosen plaintexts to actually be submitted to the web browser's connection to the SSL layer (i.e., actually performing the chosen-plaintext attack). This may potentially be accomplished through the use of a malicious plug-in. The interface to Netscape and Internet Explorer via plug-ins is standardized and readily available. Thus, it is reasonable to conceive that a plug-in containing a Trojan Horse could be devised, and furthermore that an unsuspecting user might be convinced to install such a plug-in. The prevalence of SpyWare and other known Trojan plug-ins proves this is quite feasible. In fact, the existence of so-called "SpyWare" provides for interesting comparisons. SpyWare is installed often as a hidden or poisoned plug-in, and collects data from the user without express consent. The author of the SpyWare has thus created something analogous to a Trojan Horse, rather than try to modify the browser itself to steal the user's usage habits and personal data. The prevalence of SpyWare also indicates that users can indeed be "tricked" into installing such software.

It remains to justify, however, why the present attack is easier than an attack in which a Trojan Horse is installed to simply "sniff" the user's password directly from the keyboard. The question of the comparative efficacy of capturing keystrokes as compared to performing a Chosen Plaintext Attack is not obvious. The answer lies in the challenge of writing a Trojan Horse. A modification of the web browser itself (or the underlying operating system) would be required to perform a keyboard interception. This is a substantial undertaking. First, it would require reverse-engineering or at least analyzing the source code, then making the needed modifications, and arranging for an exfiltration method. Second, the user must be persuaded to install the new version of the browser, and be convinced of its authenticity. There exist adversaries who could undertake this, but compared to the alternative in the next paragraph, this route is too difficult to be worthwhile.

On the other hand, making a Trojan Horse plug-in is much simpler. First, one would write the code from scratch and therefore no reverse-engineering is needed. Second, the volume of code that the attacking programmer would need to work with is much smaller, making this route considerably easier. Third, the (unwitting) user would only need to download a file which would take seconds or minutes, not the extended download of a new browser copy. Fourth, users install new plug-ins frequently, and both major browsers have a subsystem for adding plug-ins. This is distinct from the process of re-installing a web-browser, which requires more action on the part of the user. Installing a plug-in, by comparison, is only a few clicks.

One might imagine that a Trojan Horse plug-in could read keystrokes and thus capture (and transmit to an attacker) the user's password directly. However, this is not possible; we briefly review the reasons why. A piece of (malicious) code can see only its inputs and outputs, and perhaps some data accessible via system calls or calls from parent objects. In this case, however, the plug-in will not be granted access to form-data (which contains the user's password as typed at the keyboard) due to the manner in which plug-ins and form-data are handled by the browser. Thus, in the absence of an explicit mechanism to transfer this data to a plug-in (which is unlikely to be the case), we must assume that the plug-in can only read its specified inputs.

In Figure 1, four data paths have been marked with the letters A, B, C and D. The data which the target user is entering into the browser flows from the keyboard through the operating system via path E. The data is then entered as a form element, and then follows paths C and D through the browser to the SSL layer, and finally the TCP/IP stack en route to the Internet. A Trojan plug-in would be unable to read either Paths C and D because it is not connected to these paths. Path E is inaccessible without penetrating the operating system itself. Meanwhile, the plug-in would have free access to Path A as well as Path B (its normal write channel). In particular, the plug-in *could* be used to issue a chosen-plaintext attack via Path B into the browser, through SSL and the TCP/IP stack. Note that this is simply the "legal" mechanism by which the plug-in can transmit arbitrary plaintext messages to the SSL layer for encryption, thereby enabling a chosen-plaintext attack.

We now provide an example of how such a plug-in might be deployed. The plug-in could be part of an advertisement on a bank's web page. If the plug-in is properly written, it could be instructed to behave transparently until a certain date has passed, or until a certain user logs in. That would enable the plug-in to pass any scrutiny or testing on the part of the site administrators. The use of class-obfuscaters could result in Plug-In code that is incomprehensible to outside observers, lowering the likelihood of suspicion. Alternatively, less security conscious entities such as non-technical universities and schools, or charitable organizations with limited system administration budgets, might make easier targets.

When a link in a page connects to an object of nonstandard file-type, the user's browser will first check the list of installed plug-ins for one registered to read/display that file-type. If none is found, then the user will be prompted with a window informing him/her that a plug-in is needed before the browser can display this object. A link to the location where the plug-in can be found can be embedded into the page. In particular, the page's designer can choose data (including links) to be displayed in case loading an object fails. Since this is a novel file-type, it will certainly fail before the plug-in is downloaded. Two or three simple clicks later, the plug-in is installed and ready to be used.

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

There are two principal avenues through which this communication could occur. First, the plug-in could have access to a tcpdump-like utility, which would read these ciphertexts directly.

However, most users do not have tcpdump installed and it is unclear if a plug-in would be able to access it even if present. Ergo a method of reading these ciphertexts, which we will call a reflector, is needed.

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 daemon acting as a reflector, to capture his/her employer's SSL traffic. The plug-in or Trojan Horse 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. The daemon could masquerade as a web server, with the feedback block embedded deep in an image through the use of steganography—which would make it nearly impossible to detect through the use of an Intrusion Detection system. It would appear as though the plug-in was merely downloading an image from the reflector, now acting as a miniature web-server.

Alternatively, the reflector could simply pass the packets back "in the clear." Since both the infected user and reflector are on the same subnet, the packets would not be routed outside the Ethernet hub. Most enterprises only execute Intrusion Detection on their borders, or possibly between large sub-sectors. It would be an unusually vigilant enterprise which places Intrusion Detection Systems on each router.

Ensuring that the adversarially-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 to ensure, 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 messages from the plug-in 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.

Concatenation only occurs when two messages arrive at the SSL layer simultaneously. However, the timing characteristics of HTTPS traffic have many packets exchanged while a page is being loaded, and no packets or few between page loads, while the user is reading. Therefore a careful adversary could use delays to try to time submissions for idle periods, or simply transmit and hope for no collision. If using the latter approach, the plug-in can identify, from the absence of seeing a very short message, that concatenation has occurred, and the guess should be repeated.

4.1 Summary

The requirements listed in this section are by no means trivial. But, we believe that we have demonstrated the potential feasibility of this sort of attack.

5 Recovering "Moderately-Long" Nuggets

Here we show how even a moderately-long nugget can be easy to recover due to segmentation that occurs when the nugget falls on a block boundary. (We assume throughout this section that the data surrounding the nugget 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 nugget either by exhaustive search (in the case of Personal Identification Numbers or PINs) or by context-specific more efficient schemes (using dictionary based attacks on passwords.)

If we assume for simplicity that the nugget is chosen uniformly from a space of size S, then the expected number of guesses needed before determining the nugget is $S/2$. (Note that in the case of passwords chosen by a user, the actual entropy is likely to be much lower than would be indicated by the length of the password alone. In particular, an 8-character password typically has entropy much lower than 64 bits.) For example, a 4-digit PIN can be determined with (on average) 5,000 guesses. This certainly represents a feasible attack.

However, assume the user's PIN is 8 digits long. We claim that even in this case, we can recover the PIN with (on average) roughly this many guesses. The reason is that, with high probability, the PIN might like on a block boundary so that, say, the first 4 digits of the PIN are contained in some plaintext block P_j while the last 4 digits lie in the adjacent plaintext block P_{j+1}. Now, we can recover each half of the PIN separately at the cost of (on average) 5,000 guesses each, for a total of roughly 10,000 guesses to recover the entire PIN. This is much less than the expected $10^8/2 = 50,000,000$ guesses that one would need if the PIN were contained in a single block of plaintext.

6 Potential Solutions

We suggest two ways to prevent the attack given here. While other solutions are possible, we see no reason why one of the methods suggested below should not suffice.

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) which would anyway limit inter-operability with deployed versions of SSL.

Use (pseudo)random IVs. An immediate way to fix the vulnerability noted in this paper is to have the sender choose a new, random IV each time a new message fragment is encrypted. As we have stated several times already, this is the accepted way to encrypt using CBC. 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.

Change the mode of encryption. Another way to prevent the present attack is to use a mode of encryption other than CBC. One possibility is to use counter mode [1] with a stateful counter (note that this obviates having to choose a new, random IV each time a message is encrypted); the initial counter value can be prepended to the resulting ciphertext if desired. This has other advantages as well: the counter can simply start at 0 and therefore a random IV need not be established and shared during the negotiation phase. Furthermore, encryption can potentially be parallelized using counter mode.

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 [3, 9]. Currently, SSL does the latter rather than the former.

7 Conclusions

The attack presented here is not so easy that it can be done on the spur of the moment by the typical hacker. Yet, constructing a Trojan Horse-style plug-in to execute this attack would not be beyond the skills of someone familiar with writing plug-ins, and the problem of feedback can be likewise mitigated. Therefore, 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 examine the possibility of mandatory changes to the standard.

Acknowledgements

Thanks to 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, and Zhongchao Yu for proofreading early versions of this paper.

References

[1] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. "A Concrete Security Treatment of Symmetric Encryption: Analysis of the DES Modes of Operation." *Foundations of Computer Science*, 1997.

[2] M. Bellare, T. Kohno, and C. Namprempre. "Provably Fixing the SSH Binary Packet Protocol." *Ninth ACM Conference on Computer and Communications Security*, 2002.

[3] M. Bellare and C. Namprempre. "Authenticated Encryption: Relations among Notions and Analysis of the Generic Composition Paradigm." Asiacrypt 2000.

[4] T. Dierks and C. Allen. "RFC 2246: The TLS Protocol, Version 1.0." *Internet Engineering Task Force*, 1999.

[5] A. Freier, P. Karlton, and P. Kocher. "The SSL Protocol, Version 3.0." *Transport Layer Security Working Group Internet Draft*, 1996.

[6] S. Goldwasser and S. Micali. "Probabilistic Encryption." *JCSS*, 1984.

[7] A. Joux, G. Martinet, and F. Valette. "Blockwise-Adaptive Attackers: Revisiting the (In)Security of Some Provably Secure Encryption Models: CBC, GEM, IACBC." Crypto 2002.

[8] C. Kaufmann, R. Perlman, and M. Speciner. *Network Security: Private Communication in a Public World*, second edition, Prentice Hall, 2002.

[9] H. Krawczyk. "The Order of Encryption and Authentication for Protecting Communications (or: How Secure Is SSL?)." Crypto 2001.