# Here Come The $\oplus$ Ninjas

Thai Duong Juliano Rizzo

May 13, 2011

**Abstract**

This paper introduces a fast blockwise chosen-plaintext attack against SSL 3.0 and TLS 1.0. We also describe one application of the attack that allows an attacker to efficiently decrypt and obtain authentication tokens embedded in HTTPS requests[1] The resulting exploits work for major web browsers at the time of writing.

## 1 Introduction

The Secure Sockets Layer (SSL) protocol[2] is widely-used for securing communication over the Internet. The SSL standard allows for symmetric-key encryption using either block ciphers or stream ciphers. Implementations usually utilize block ciphers, and the attack in this paper applies only when block ciphers are used. The SSL standard mandates the use of the CBC mode encryption [8] with chained initialization vectors (IVs); i.e., the IV used when encrypting a message should be the last block of the previous ciphertext. Unfortunately, CBC mode encryption with chained IVs is insecure [6, 17], and this insecurity extends to SSL [1, 2, 15]. We show that this vulnerability enables a man-in-the-middle attacker mounting a chosen-plaintext attack against SSL to quickly recover plaintext. In case of HTTPS [16], we demonstrate that an attacker can decrypt and obtain authentication tokens such as HTTP cookie [9, 14].

**Contributions** This paper describes a new plaintext-recovery attack against real world implementations of the CBC encryption scheme. The refined blockwise attack model introduced in this paper better captures SSL attackers than any other chosen-plaintext attack models. As a consequence, the new attacks described here are more dangerous, generic and efficient than any previous published results involving chosen-plaintext attack against SSL. As a proof of concept, this paper also present efficient browser exploits that actually decrypt HTTPS requests to obtain HTTP cookies.

## 2 Related Work

The chosen-plaintext attacks described in this paper are based on a known attack against SSL with chained IVs that was reported in [1, 2, 5, 6, 15]. In SSL the plaintext is fragmented into records of length less than or equal to $2^{14}$ bytes. Each record is then encrypted in CBC mode with chained IVs; i.e., the CBC IV for each record except the first is the previous records' last ciphertext block. Therefore, the IV is predictable and an attacker intercepting network traffic will know the IV for the next record to be encrypted before the next record is actually encrypted. This means that if an attacker can control the first block of the input into SSL's underlying CBC encryption scheme, he will be able to control the corresponding input to the underlying block cipher. Since a block cipher is deterministic, an attacker could use this to glean information about a previously encrypted message (by looking to see if some value was ever the input to a previous blocks cipher invocation). As described by W. Dai in [6] the attack itself is very simple. Remember that in CBC mode, each plaintext block is XOR'ed with the last ciphertext block and then encrypted to produce the next ciphertext block. Suppose the attacker suspects that plaintext block $P_i$ in some previously seen SSL record might be $x$, and wants to test whether that is the case, he would choose the first plaintext block of the next record $P_j$ to be $C_{j-1} \oplus C_{i-1} \oplus x$. If his guess is correct, then $C_j = E_k(P_j \oplus C_{j-1}) = E_k(P_i \oplus C_{i-1}) = C_i$, and so he can confirm his guess by looking at whether $C_j = C_i$.

[1] Besides HTTP over SSL, we also target WebSocket over SSL. For convenience, we simply refer to these generically as "HTTPS."

[2] The attack described here applies to SSL 3.0 [11] and TLS 1.0 [7]. For convenience, we simply refer to these generically as "SSL."

As observed in [1], at its core the attack just described is an example of what is called a "blockwise" attack [5, 13]. In contrast to a "standard" chosen-plaintext attack where messages are viewed as atomic, in a blockwise attack an adversary is assumed to have the additional ability to insert plaintext blocks within some longer message as that message is being encrypted [13]. Although Dai's attack does not exactly follow this paradigm, one may cast that attack in this light due to the fact that the IV for each record is taken to be the final block of the ciphertext corresponding to the preceding record. 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 attack feasible.

Essentially the Dai's attack has been used previously to attack SSH [5]. In [1,2], G. Bard extended the attack to SSL. It was both surprising and helpful for the authors of this paper to discover G. Bard's work. Indeed in 2004 and later in 2006, G. Bard already discussed the scenario, requirements, possible implementations, and solutions of using Dai's method to attack SSL , which is basically what we independently rediscover in 2011. Nevertheless, it is important to stress that while [1, 2] provided a solid background on chosen-plaintext attacks against SSL with chained IVs, they did not show any practical exploit that can break real world implementations of SSL. That also explains why the world has ignored the warning of SSL's insecurity in those papers, and kept using SSL 3.0 and TLS 1.0. That said, in order to avoid repeating what had been already done, we only focus on the new ideas in this paper, and refer as much as possible to [1,2] and other papers for other relevant details.

# 3 Chaining of Predictable IVs

In order to mount the Dai's attack, the adversary has to control the entire first block $P_j$.[3] Usually, in practice, some header data is prepended to the plaintext before encryption, i.e., some bits of $P_j$ are not controllable by the adversary. This means that an attacker may not be able to force a user's underlying CBC scheme to encrypt the block $C_{j-1} \oplus C_{i-1} \oplus x$. As analyzed in [5], an attacker will, however, be able to mount Dai's attack if $C_{i-1}$ and $C_{j-1}$ are identical in the bits that the attacker cannot control in $P_j$. However, the success probability of the attack is significant reduced, because the attacker now has to wait for a collision on some bits that he has no control. This section outlines a technique that allows an adversary to mount the Dai's attack with a success probability that can approach 100% even if it cannot control the entire $P_j$.

Recall that $C_{j-1}$, which is known to the adversary, is used as the IV to encrypt $P_j$. The idea is to use $C_{j-1}$ to predict $C_j$. Since $C_j$ is again used as the IV to encrypt $P_{j+1}$, knowing $C_j$ in advance basically allows the adversary to use $P_{j+1}$, which he fully controls, to guess for $P_i$. To simplify the discussion, suppose that the adversary cannot control the first byte of $P_j$, e.g., a magic byte or length field, denotes as $w$, is prepended to the plaintext block before it is actually encrypted. Suppose there is a lookup table $T$ that maps $C_{j-1}[1]$, i.e., the first byte of $C_{j-1}$, to a fixed pair of $(P^*, C^* \leftarrow E_k(P^*))$ such that $w = C_{j-1}[1] \oplus P^*[1]$. The adversary computes $P_j \leftarrow C_{j-1} \oplus P^*$ and $P_{j+1} \leftarrow C^* \oplus C_{i-1} \oplus x$. Basically he ensures that the first byte of $P_j$ is $w$ and use $P_{j+1}$ to make a guess for $P_i$. If his guess is correct, then we can check that:

$$\begin{aligned} C_{j+1} &= E_k(C_j \oplus P_{j+1}) \\ &= E_k(E_k(C_{j-1} \oplus P_j) \oplus P_{j+1}) \\ &= E_k(E_k(C_{j-1} \oplus C_{j-1} \oplus P^*) \oplus P_{j+1}) \\ &= E_k(E_k(P^*) \oplus P_{j+1}) \\ &= E_k(C^* \oplus C^* \oplus C_{i-1} \oplus P_i) \\ &= C_i. \end{aligned}$$

Since the adversary is performing a chosen-plaintext attack, he can build such a table $T$ easily. The adversary just needs to choose $P^*$ and store the corresponding $C^*$ returned by the encryption oracle. If the adversary cannot control $n$ bits of $P_j$, then $T$ would contain at most $2^n$ entries. He even does not need to build a full table before starting to guess for $P_i$, but he can build in on the fly, i.e., if $T$ contains an entry for the current IV, then make a guess for $P_i$; otherwise add a new entry to $T$, encrypt a random block $P^*$, and lookup the table for the new IV again.

[3] We use the same notation as in the description of the Dai's attack above.

# 4 The Blockwise Attack

In cryptography, security notions are usually defined by combining a security goal and an attack model. The more accurate an attack model captures attackers in the real world, the better the security notions are. The blockwise attack (BA) model [5, 13] was proposed with that motivation. [10] explained why the BA model better captures real world attackers than the standard one as follows

> The standard model for the chosen-plaintext attack is message oriented: i.e. the messages are viewed as atomic object which cannot be split into blocks. Thus, adversaries can only be adaptive between the messages. However, sometimes the encryption process has to be started even if the entire plaintext is not known. For example, in real-time applications, the cryptographic device cannot store the whole plaintext before the starting of the encryption. [...]Moreover, in many practical applications, cryptographic devices (smart cards) are memory restricted. Then, if messages are too large, they cannot be stored in the cryptographic module before the beginning of the encryption process. Therefore, the message must be sent block by block to the cryptographic module which returns on-the-fly the output block $C[i]$, say just after the query of the input block $M[i]$ in some implementations. As a consequence, the adversary model needs to be changed to take into account attackers querying messages block by block. In the BA model, attackers are more adaptive than standard adversaries: they are adaptive during the encryption query, i.e. between each block of messages, and not only between the encryption queries, i.e. between the messages. Hence the name of "blockwise" adversaries.

It is known that the BA model is stronger than the standard one. [4] proved that the CBC encryption scheme is secure in the standard model. However, in [13], Joux, Martinet and Valette used the same idea as Dai's attack to show that the CBC encryption scheme is not secure in the BA model after only two encrypted blocks. It is worth noticing the attack against CBC in [13] invalidates the security proof by building distinguisher but do not allow to recover the secret key or to totally break the scheme.

## 4.1 A Refinement of The Blockwise Attack

We now present blockwise chosen-boundary attack (BCBA), which is a refinement of the blockwise attack. Suppose the encryption mode in use is CBC. Let $b$ be the block size in byte, and let $m$ be a (padded) message consists of $l$ bytes $m[1], m[2], ..., m[l]$.[4] $m$ would be divided into $s$ $b$-byte blocks $p_1, p_2, ..., p_s$. Consider the block boundary, denoted as $x$, between $p_1$ and $p_2$. For now $x$ is at the position between $m[b]$ and $m[b+1]$. In the BCBA model, it is assumed that before $m$ is actually encrypted, an attacker can move block boundaries to any position at his will, e.g., he can move $x$ to the position between $m[1]$ and $m[2]$. One way for an attacker to control the position of block boundaries is to prepend string of arbitrary length to $m$, e.g., if an attacker can prepend $r$ bytes to $m$ such that $r < b$, then all block boundaries in $m$ would be shifted $r$ positions to the left. Figure 1 illustrates a concrete example of what is just described. In this case the block size in bytes is 8 and $m$ consists of 9 bytes (or 16 bytes including padding.) By prepending an arbitrary 7-byte string to $m$, an attacker shifts the block boundary 7 positions to the left. In general, for any consecutive pair of bytes $m[i]$ and $m[i+1]$, an attacker can always prepend less than $b$ bytes to $m$ such that there is a block boundary between $m[i]$ and $m[i+1]$. Therefore, an attacker can choose which position to become a block boundary. Hence the name of "chosen-boundary" attackers.

## 4.2 Plaintext-Recovery Attack Against CBC Mode

In [4], Bellare et al. have proved that the CBC encryption scheme is secure in the standard model up to the encryption of $2^{n/2}$ blocks, where $n$ denotes the block length in bits of a block cipher. in [13], Joux, Martinet and Valette used the same idea as Dai's attack to show that the CBC encryption scheme is not secure in the BA model after only two encrypted blocks. In this section, we briefly recall the CBC encryption mode and then we describe how to mount a plaintext-recovery attack against CBC in the BCBA model.

Let $E_k$ be a block cipher with secret key $k$ and block-size $b$ bytes and let $m$ be the (padded) message to encrypt that consists of $l$ bytes $m[1], m[2], ..., m[l]$. $m$ is divided into $n$ $b$-byte blocks denoted by $(P_1, ..., P_n)$. A

[4] The block size and padding scheme in use do not affect the attacks described in this paper.

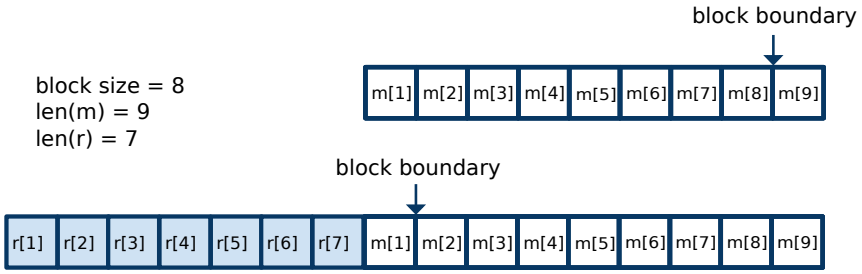


Figure 1: Prepending string to a message allows an attacker to choose block boundaries.

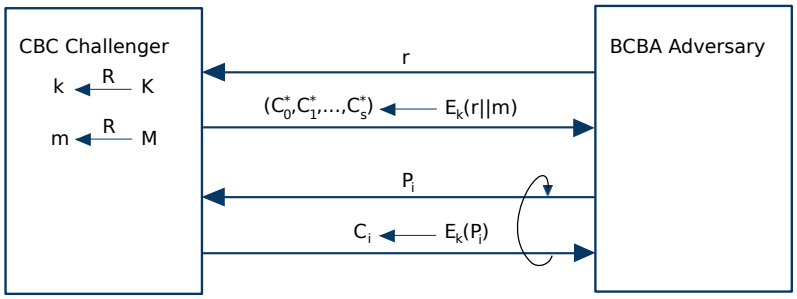


Figure 2: Blockwise chosen-boundary adversary against CBC mode.

random $b$-byte initial value $IV$ is generated by the encryption box. The CBC mode of encryption with random initial value is defined as follows:

$$C_0 = IV;$$
$$C_i = E_k(P_i \oplus C_{i-1}) \qquad \text{for i=1,...,n.}$$

The transmitted ciphertext is $(C_0, C_1, ..., C_n)$.

If $C_{i-1}$ is known when choosing $P_i$, an attacker can easily mount the Dai's attack to break the security proof of CBC in [4]. Now consider the security game illustrated in Figure 2. Like other security games, this game consists of a challenger and an adversary. The challenger is a CBC encryption oracle that on startup generates a random key $k \in K$ and choose a random message $m$ in the message space $M$. On the other hand, in addition to being adaptive from one block to the next within a single message, the adversary is also allowed to prepend string of arbitrary length to $m$ before $m$ is actually encrypted. In order to give the adversary that flexibility, the security game is designed to consist of two phases:

**Phase 1 (Chosen-Boundary)** The adversary generates a random string $r$, and send $r$ to the challenger. The challenger chooses a random $IV$, prepends $r$ to $m$, encrypts the resulting (padded) string in CBC mode under key $k$ and the random $IV$ (denoted as $C_0^*$), and sends the ciphertext $(C_0^*, C_1^*, ..., C_s^*)$ to the adversary.

**Phase 2 (Blockwise)** The adversary chooses a block $P_i$, sends it to the challenger. The challenger encrypts $P_i$ using key $k$ and the last ciphertext block as IV, and sends the ciphertext $C_i$ to the adversary. The adversary repeats this phase again until it wants to stop.

The goal of the adversary is to decrypt $m$.

## First Byte Decryption

It turns out that there is a deterministic algorithm that allows the adversary to obtain $m[1]$ after as many as 256 repetitions of phase 2. In other words, after querying the challenger 256 queries in phase 2, the adversary can decrypt the first byte of $m$. If the adversary is lucky (with probability $1/256$), it can obtain $m[1]$ with the first query. On average, it has to try 128 queries. The algorithm proceeds as follows:

**Step 1** Generate a random string $r$ that consists of $b-1$ bytes, and send $r$ to the challenger.

**Step 2** The challenger prepends $r$ to $m$ to obtain $P^*$. $P^*$ would be divided into $s$ $b$-byte blocks, denoted as $P_1^*, ..., P_s^*$. Since $r$ consists of $b-1$ bytes, the first plaintext block $P_1^*$ would be equal to $r||m[1]$. The challenger encrypts $P^*$ and transmits the resulting ciphertext $(C_0^*, C_1^*, ..., C_s^*)$ to the adversary. The goal of the adversary is now to guess the value of $m[1]$. Let $i = 0$.

**Step 3** Let $IV$ be the last ciphertext block obtained from the challenger, e.g., $IV$ is $C_s^*$ after step 2. Send $P_i = C_0^* \oplus IV \oplus (r||i)$ to the challenger.

**Step 4** Receive $C_i$ from the challenger.

**Step 5** If $C_i = C_1^*$, output $i$; otherwise, increase $i$ and goes back to step 3.

We claim that the above algorithm would terminate and output $m[1]$ after a maximum of 256 repetitions of the last three steps. Actually the last three steps of the algorithm are similar to the Dai's attack, i.e., each execution makes a guess for the correct value of some block, which in this case is $P_1^*$. The key idea here is that the first two steps (the chosen-boundary phase) make sure that all but the last byte of $P_1^*$ are known. Thus, an attacker needs to guess for only one byte instead of a whole block, and that allows a fast recovery of $m[1]$. Indeed since $i$ is increased after each failed guess, $i$ would be eventually equal to $m[1]$. Then we can check that:

$$\begin{aligned} C_i &= E_k(IV \oplus P_i) \\ &= E_k(IV \oplus C_0^* \oplus IV \oplus (r||i)) \\ &= E_k(C_0^* \oplus (r||m[1])) \\ &= E_k(C_0^* \oplus P_1^*) \\ &= C_1^*. \end{aligned}$$

It is easy to see that when $i \neq m[1]$, $C_i$ is different from $C_1^*$. As a consequence, the attacker can easily find $m[i]$ after trying on average 128 queries.

### Full Message Decryption

It is straight-forward to extend the first byte decryption algorithm to a full message decryption algorithm. Suppose the attacker has obtained $m[1], m[2], ..., m[i-1]$, and he now wants to obtain $m[i]$. Recall that in Section 4.1 we showed that for any consecutive pair of bytes $m[i]$ and $m[i+1]$, an attacker can always prepend less than $b$ bytes to $m$ such that there is a block boundary between $m[i]$ and $m[i+1]$. In other words, an attacker can always make $m[i]$ become the last and only unknown byte of some block, and he then uses the first byte decryption algorithm to obtain $m[i]$. For a $l$-byte message $m$, an attacker needs to query the challenger on average $128 * l$ times to obtain $m$. It is worth noticing that the cost analysis stated here is for the generic case when bytes in $m$ is random, i.e., each of them can take any value between $[0..255]$. If bytes in $m$ take only values in some smaller range, then the cost is smaller.

# 5 Application: Decrypting HTTPS Requests

## 5.1 HTTPS Overview

The HTTPS protocol is a combination of the HTTP [9] with the SSL protocol to provide encrypted communication and secure identification of a network web server. HTTPS connections are often used for payment transactions on the World Wide Web and for sensitive transactions in corporate information systems. HTTP operates at the highest layer of the OSI Model [19], the Application layer; but the security protocol operates at a lower sublayer, encrypting an HTTP message prior to transmission and decrypting a message upon arrival. Strictly speaking, HTTPS is not a separate protocol, but refers to use of ordinary HTTP over an encrypted SSL connection. A HTTP request message consists of the following:

- Request line consisting of a method, resource path and HTTP version, such as `GET /logo.png HTTP/1.1`, which requests a resource called `/logo.png` from server.
- Headers, such as `Cookie: sessionid=1cf1e8dacc3b29c2fc9161baf30539fd;`.
- An empty line.
- An optional message body.

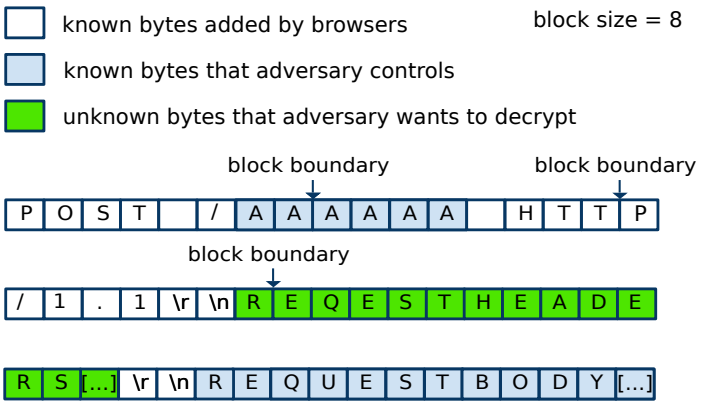


Figure 3: Performing a blockwise chosen-boundary attack against HTTPS with POST requests.

The request line and headers must all end with <CR><LF> (that is, a carriage return followed by a line feed.) The empty line must consist of only <CR><LF> and no other whitespace.

In HTTPS, everything in the HTTP message is encrypted, including the headers, and the request/response load. SSL receives the HTTP message from the Application Layer as raw data. This message is fragmented into blocks of length less than or equal to $2^{14}$ bytes. These blocks are optionally compressed [5] and are then encrypted, then the resulting ciphertext blocks are sent to server. The most important information inside a HTTP request is usually the cookie header. Since HTTP is a stateless protocol, a HTTP cookie, also known as a web cookie, or browser cookie, is used for an origin website to send state information to a user's browser and for the browser to return the state information to the origin site. The state information can be used for authentication, identification of a user session, user's preferences, shopping cart contents, or anything else that can be accomplished through storing text data. Most web sites use cookies as the only identifiers for user sessions. If a web site uses cookies as session identifiers, attackers can impersonate users' requests by stealing a full set of victims' cookies. From the web server's point of view, a request from an attacker has the same authentication as the victim's requests; thus the request is performed on behalf of the victim's session.

Giving the important role of HTTP cookies, there are several cookie stealing attacks as well as countermeasures have been proposed [18]. Actually HTTPS is mainly used to protect the cookies from being stolen by network eavesdropping attacks. A server can specify the `secure` flag while setting a cookie, which will cause the browser to send the cookie only over an encrypted SSL channel. The attacks described in this paper allow an attacker to obtain the cookies even if HTTPS is in use and the `secure` flag is turned on. The novelty of our attacks lie in the fact that they are the first attacks that actually decrypt HTTPS requests by exploiting cryptographic weaknesses of using HTTP over SSL.

## 5.2 Threat Model

Suppose that Alice uses her browser to access Bob's web server over HTTPS at `https://bob.com`. Bob's web server sets some session identifer cookies inside Alice's browser so that it can authenticate subsequent requests from Alice. Alice then visits `http://mallory.com` where Mallory has installed some malicious code that allows Mallory to decrypt Alice's requests sent to `https://bob.com` and finally obtain her cookies. We make three assumptions about the capability of Mallory:

**Network Eavesdropping Privilege** Mallory can capture encrypted HTTPS requests sent by Alice. As noted by G. Bard, since the ciphertext blocks travel over the Internet, 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).

**Chosen-Boundary Privilege** Mallory can force Alice's browser to make arbitrary cookie-bearing requests to Bob's web server over HTTPS. Moreover, Mallory can also control the resource path of these requests, i.e., `/logo.png` as shown above. Note that this capability basically allows Mallory to control block boundaries of the targeted request headers.

[5] The attacks in this paper do not apply directly when compression is used; however, compression is rarely used in practice.

**Blockwise Privilege** After making Alice's browser open some HTTPS requests to the server, Mallory can append arbitrary plaintext blocks to each ongoing request. Note that this capability together with the first capability basically allows Mallory to perform the Dai's attack against HTTPS.

In the next section, we show how Mallory can gain the last two privileges by leveraging browser features or plugins. We now show that with these assumptions, Mallory can decrypt and obtain the request headers which contain the cookies he wants. One can see that Mallory now becomes the BCBA adversary and Alice's browser becomes the CBC challenger, as part of the security game in Figure 2. For the chosen-boundary phase, $r$ is the resource path which is controlled by Mallory and the secret message $m$ consists of Alice's cookies and other headers. For the blockwise phase, the plaintext blocks that Mallory appends to each ongoing request are $P_i$; Mallory receives $C_i$ by capturing Alice's encrypted HTTPS requests. Figure 3 shows how Mallory can mount a blockwise chosen-boundary attack against Alice's browser to decrypt the first byte of the targeted request header. Suppose the block size in bytes is 8, the attack proceeds as follows:

**Step 1** Mallory forces Alice's browser to open a HTTPS POST request to `http://bob.com/AAAAAA`. Alice's browser uses SSL to derive a shared secret key $k$ with Bob's web server, then it utilizes a block cipher in CBC mode to compute $C_1|C_2|C_3|...|C_n \leftarrow E_k(P)$ where $P$ is `POST /AAAAAA HTTP/1.1<CR><LF><REQUEST HEADERS><CR><LF><REQUEST BODY>`, and sends $C_1|C_2|C_3|...|C_n$ to server.

**Step 2** Mallory captures $C_1, C_2, C_3, ..., C_n$. Note that $C_3$ is the encryption of $P_3 =$ `P/1.1<CR><LF><X>`, i.e., the first 7 bytes are known to Mallory and the last byte, denotes as $X$, is the first byte of the request headers. Mallory's goal now is to obtain $X$. Let $W[1..l]$ is the set of allowed bytes in HTTP header and let $i = 1$.

**Step 3** Let $IV$ be the last ciphertext block that Mallory captures, e.g., $IV$ is $C_n$ after step 2. Mallory computes $P_i = IV \oplus C_2 \oplus P_{guess}$, where $P_{guess} =$ `P/1.1<CR><LF><W[i]>`. Mallory appends $P_i$ to the existing request, e.g., $P_i$ is appended to the request body (this is the last assumption.) Alice's browser would compute $C_i \leftarrow E_k(IV \oplus P_i)$ and send $C_i$ to server.

**Step 4** Mallory captures $C_i$. If $C_i = C_3$, he knows that $X$ is equal to $W[i]$. Otherwise he increases $i$ and goes back to step 3.

We claim that the above attack would terminate and Mallory would obtain the first byte of the request header after a maximum of $|W|$ repetitions of the last two steps. Indeed what Mallory does is exactly the same as the first byte decryption algorithm described in Section 4.2. So once again the same argument stated in Section 4.2 can be applied, and one can also see that it is straight forward for Mallory to extend the attack to decrypt the whole request headers and obtain Alice's cookies. In practice, most bytes of the request headers are known, so Mallory can optimize the attack by decrypting only the cookie header.

The remaining question now is if there is any real world adversary that actually has the same privileges as Mallory. As described in the first paragraph of this section, we assume that Alice visits `http://mallory.com` where Mallory has installed some malicious code, which we call "Mallory's agent" or simply "the agent" hereinafter. The agent can be regarded as a browser exploit that leverages browser features or plugins to help Mallory gain his desired privileges. In addition, any real world Mallory must also consist of a network sniffer that intercepts SSL traffic, and sends ciphertext blocks to the agent. The agent can communicate with the network sniffer via basic socket programming, with synchronization done using timing or message length (refer to [1, 2] for a discussion.) In the next section, we present a "portfolio" of browser technologies that can be used to implement Mallory's agent.

## 5.3 From Browser Technologies To Mallory's Privileges

Recall that the necessary condition of Mallory's agent is the ability to send *cookie-bearing* requests to Bob's server, since Alice's browser cookies are what Mallory is going after. The author of this papers then collected a list of browser features and plugins that satisfy this condition. Surprisingly, there are a lot of them, including but not limited to Javascript XMLHttpRequest API, HTML5 WebSocket API, Flash URLRequest API, Java Applet URLConnection API, and Silverlight WebClient API (refer to [18] for an excellent discussion of this topic.) Actually Mallory can make Alice's browser open arbitrary cookie-bearing requests to Bob's server using techniques similar to cross-site request forgery attacks [3]. For example, Alice might be browsing a chat forum at `http://mallory.com` where Mallory has posted a message. Suppose that Mallory has crafted an HTML image element that references a resource on Bob's server, e.g., `<img src="https://bob.com/AAAAAA">`, then Alice's browser would send a cookie-bearing HTTPS request to `https://bob.com/AAAAAA`. So it is easy for Mallory to gain the chosen-boundary privilege. This approach does not allow Mallory to gain the blockwise

privilege though. There are some reasons. First, web browsers tend to open a new SSL connection to server for each HTTP request. Secondly, even if Mallory can make web browsers re-use the same SSL connection for multiple HTTP requests, Mallory cannot control the first few bytes of each request because they are always set as a fixed string such as `GET /`, `POST /`, etc. We note that these strings are too long so that the technique described in Section 3 takes too much time to run. Finally, most browsers do not accept arbitrary strings as resource path [18]. Nevertheless, we eventually realize what Mallory really needs is to make Alice's browser open bi-directional communication channels to server. We use this condition to further narrow down the initial list, and what left are HTML5 WebSocket API, Java URLConnection API, and Silverlight WebClient API. We stress that this list is by no means complete. There might be other features or plugins that as "good" as the ones listed here but we did not know. At the end of the day, we just want to make our point that there is a vulnerability in HTTPS, and it is just a matter of browser features for adversaries to implement efficient exploits. In appendix A we describe three browser exploits that can act as Mallory's agent. The existence of these exploits confirm the existence and practicability of blockwise chosen-boundary attacks in general and the SSL vulnerability in particular.

# 6 Impact

# 7 Countermeasures

**Acknowledgements**

# References

[1] G.V. Bard. The Vulnerability of SSL to Chosen-Plaintext Attack. *Cryptology ePrint Archive, Report 2004/111*, 2004.

[2] G.V. Bard. A Challenging but Feasible Blockwise-Adaptive Chosen-Plaintext Attack on SSL. *SECRYPT*, pages 7–10, 2006.

[3] A. Barth, C. Jackson, and J.C. Mitchell. Robust Defenses for Cross-site Request Forgery. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 75–88. ACM, 2008.

[4] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A Concrete Security Treatment of Symmetric Encryption. In *FOCS*, page 394. Published by the IEEE Computer Society, 1997.

[5] M. Bellare, T. Kohno, and C. Namprempre. Authenticated Encryption in SSH: Provably Fixing the SSH Binary Packet Protocol. In *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pages 1–11. ACM, 2002.

[6] W. Dai. An Attack Against SSH2 Protocol, Feb. 2002. *Email to the ietf-ssh@ netbsd. org email list*, 2002.

[7] T. Dierks. The TLS Protocol Version 1.0 – RFC 2246. *IETF Request For Comments*, 1999.

[8] M. Dworkin. Recommendation for Block Cipher Modes of Operation: Methods and Techniques. Technical report, DTIC Document, 2001.

[9] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol–HTTP/1.1. Technical report, RFC 2616, June, 1999.

[10] P.A. Fouque, A. Joux, and G. Poupard. Blockwise Adversarial Model for On-line Ciphers and Symmetric Encryption Schemes. In *Selected Areas in Cryptography*, pages 212–226. Springer, 2005.

[11] A. Frier, P. Karlton, and P. Kocher. The SSL 3.0 Protocol. *Netscape Communications Corp*, 18:2780, 1996.

[12] C. Jackson, A. Barth, A. Bortz, W. Shao, and D. Boneh. Protecting Browsers From DNS Rebinding Attacks. *ACM Transactions on the Web (TWEB)*, 3(1):1–26, 2009.

[13] A. Joux, G. Martinet, and F. Valette. Blockwise-Adaptive Attackers Revisiting The (In) Security of Some Provably Secure Encryption Modes: CBC, GEM, IACBC. *Advances in Cryptology - CRYPTO 2002*, pages 231–248, 2002.

[14] D. Kristol and L. Montulli. HTTP State Management Mechanism – RFC 2965. *RFC Editor United States*, 2000.

[15] B. Moeller. Security of CBC Ciphersuites in SSL/TLS: Problems and Countermeasures. Available at http://www.openssl.org/~bodo/tls-cbc.txt, 2004.

[16] E. Rescorla. HTTP over TLS – RFC 2818. *RFC Editor United States*, 2000.

[17] P. Rogaway. Problems with Proposed IP Cryptography. Available at http://www.cs.ucdavis.edu/~rogaway/papers/draft-rogaway-ipsec-comments-00.txt, 1996.

[18] M. Zalewski. Browser Security Handbook. Available at http://code.google.com/p/browsersec/, 2010.

[19] H. Zimmermann. OSI Reference Model–The ISO Model of Architecture for Open Systems Interconnection. *IEEE Transactions on Communications*, 28(4):425–432, 1980.

# A Browser Exploit Implementations

Recall that Alice visits http://mallory.com where Mallory has installed an agent that allows him to send cookie-bearing requests to https://bob.com. People familiar with browser security probably have noticed that the same-origin security policy in modern browsers may prevent Mallory's agent from running. As described by Zalewski in [18], the principal intent for this same-origin mechanism is to make it possible for largely unrestrained scripting and other interactions between pages served as a part of the same site (understood as having a particular DNS host name, or part thereof), whilst almost completely preventing any interference between unrelated sites. In practice, however, there is no single same-origin policy, but rather, a set of mechanisms with some superficial resemblance, but quite a few important differences. We observe that in some circumstances, Mallory can leverage these differences so that his agent running inside http://mallory.com can still send requests to https://bob.com. For example, in Java, it is widely known that if mallory.com and bob.com share the same IP address[6], then they are considered same-origin.[7] In the WebSocket protocol, JavaScript programs in mallory.com can still access WebSocket services in bob.com if the services are configured to allow connections from mallory.com. To put it differently, all it takes for Mallory's agent to run is the ability to insert JavaScript code into any website that is allowed to access WebSocket services in bob.com. There is a large class of attacks known as cross-site scripting that makes the just described scenario possible. Therefore, in order to focus on cryptographic aspects of this work, we assume that Mallory's agent can send requests to https://bob.com. That said, we acknowledge that given the current state of browser security, the ability to send same-origin requests from different DNS host names may open other non-cryptographic attacks which are, however, out of scope of this discussion.

## A.1 Java Applet Exploit

## A.2 Silverlight Exploit

## A.3 HTML5 WebSocket Exploit

The current version of the WebSocket protocol in most browsers is implemented based on what is known as version 76 of the WebSocket specification.[8] Suppose there is a WebSocket service at bob.com/websocket. Mallory can connect to that service over SSL using the following JavaScript code:

```
var s = new WebSocket("wss://bob.com/websocket?ABCDEF");
s.onopen = function(e) {
  console.log("opened");
  s.send("Hello, world!");
  s.send("Here come the + ninjas");
}
```

The browser would open a SSL connection to bob.com, and perform the initial WebSocket handshake. We stress that the cookie header is included in the handshake requests. In order to gain the chosen-boundary privilege, Mallory can append bytes to the query string of the destination URL, e.g., ABCDEF as shown above.[9] For each

[6] Mallory can use DNS re-pinning attacks [12] to trick browsers into believing that mallory.com has the same IP address as bob.com.

[7] http://download.oracle.com/javase/6/docs/api/java/net/URL.html.

[8] http://tools.ietf.org/html/draft-hixie-thewebsocketprotocol-76.

[9] This query string would be ignored by the targeted WebSocket service

call to the `send` API function, the browser would send a new SSL record (in the on-going SSL connection) containing the data passing to `send`. Mallory can use this API function to gain the blockwise privilege. There is one minor issue though. WebSocket packs data into frames. Although a frame can contain binary or UTF-8 data, current version of the `send` function in most browsers does not support binary frames. Each frame of UTF-8 data starts with a 0x00 byte and ends with a 0xFF byte, with the UTF-8 text in between. This data framing technique creates two problems. First, due to the prepended 0x00 byte, Mallory cannot control the entire first block. Fortunately, Mallory can use the technique described in section 3 to bypass this limitation. Secondly, all data blocks sent by Mallory must be valid UTF-8 string; otherwise browsers would raise an invalid data type error. Recall that in the chaining of predictable IVs technique, Mallory needs to send two data blocks, denotes as $P_1$ and $P_2$, i.e., the first block is used to set the IV for the second block, which in turn is used to make a guess for the targeted plaintext block. While Mallory cannot control $P_1$, since its value depends on a random IV, he can control the $P_2$ (refer to section 3 for detailed explanation.) In other words, Mallory can deploy a randomized algorithm such that if the last ciphertext block of the last SSL record, i.e., the IV of the next record, makes $P_1$ decode as valid UTF-8 string, then he makes a guess in $P_2$; he simply tries again otherwise.