Vulnerabilities in standard cryptographic protocols

BEAST Vulnerability in SSL/TLS

Arshdeep Singh | 2016CS50625

BEAST (Browser Exploit Against SSL/TLS) allows man in the middle attacker to uncover information from an encrypted SSL/TLS1.0 session.

1 PROTOCOL EXPLANATION

Secure Sockets Layer (SSL) and Transport Layer Security (TLS) are cryptographic security protocols. They are used to make sure that network communication is secure. Their main goals are to provide data integrity and communication privacy. The SSL protocol was the first protocol designed for this purpose and TLS is its successor. SSL is now considered obsolete and insecure, so modern browsers such as Chrome or Firefox use TLS instead. SSL and TLS are commonly used by web browsers to protect connections between web applications and web servers.

SSL and TLS at its core uses block ciphers which are symmetric encryption methods and the key is settled before using Asymmetric encryption also referred to as Public Key Cryptography.

Block Ciphers: If you use a block cipher, data is split into fixed-length blocks (e.g. 64-bit or 128-bit blocks) and then encrypted. If the last block of data is shorter than the specified block length, the algorithm uses padding to fill the empty space. Blocks are usually padded using random data. Popular block ciphers include AES, Blowfish, 3DES, DES, and RC5.

Padding: Block ciphers have a specified fixed length and most of them require that the input data is a multiple of their size. It is common that the last block contains data that does not meet this requirement. In this case, padding (usually random data) is used to bring it to the required block length.

Initialization Vector (IV): An initialization vector is a random (or pseudorandom) fixed-size input used in encryption methods. The main purpose of an IV is starting off an encryption method. In cipher modes like Cipher Block Chaining (CBC) each block is XOR-ed with the previous block. In the first block, there is no previous block to XOR with. An Initialization Vector is used as an input to the first block to start off the process.

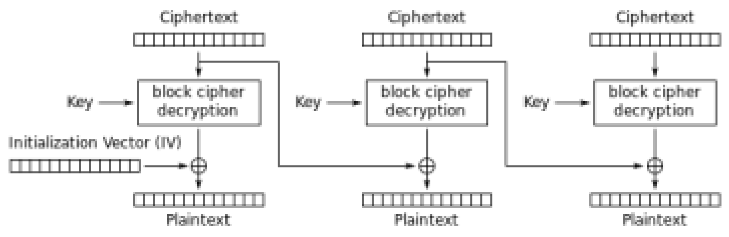
Cipher Block Chaining (CBC): When using CBC, each block is XORed with the previous ciphertext before encryption. This eliminates the problem of repeating patterns. An IV is needed to encrypt the first plaintext block. Parallel processing is not possible since the blocks are chained. There is one major disadvantage of CBC: if part of the message is garbled or lost, the remainder of the message is lost

In SSL and TLS block of message is appended with HMAC (hash-based message authentication code) calculated using the key established earlier. Then it is padded in order to make it usable by block ciphers using CBC mode. Below you can find the encryption and decryption using block cipher in CBC mode The plaintext is divided into blocks regarding the encryption algorithm (AES, DES, 3DES) and the length is a multiple of 8 or 16. If the plaintext doesn't fill the length, a padding is added at the end to complete the missing space.

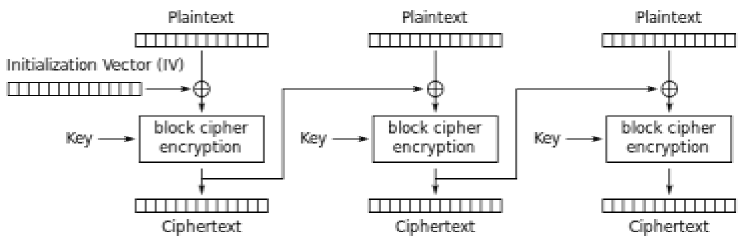
Encryption: $C_i = E_k(P_i \oplus C_{i-1})$, and $C_0 = IV$

Decryption: $P_i = D_k(C_i) \oplus C_{i-1}$, and $C_0 = IV$

Where P_i and C_i denote i-th block in plaintext and ciphertext respectively. E_k and D_k are the encryption and decryption algorithm.



Cipher Block Chaining (CBC) mode decryption



Cipher Block Chaining (CBC) mode encryption

2 VULNERABILITY EXPLANATION

The ability to mount an adaptive chosen plaintext attack with predictable initialization vectors (IVs) against SSL/TLS using cipher block chaining (CBC) was known in 2004. but until late 2011 was thought to be largely theoretical. Researchers Thai Doung and Juliano Rizzo found a way to exploit the vulnerability and demonstrated a live attack. Although the vulnerability is cryptographic in nature, it requires certain conditions to be successful

BEAST leverages a type of cryptographic attack called a **chosen-plaintext attack**. The attacker mounts the attack by choosing a guess for the plaintext that is associated with a known ciphertext. To check if a guess is correct, the attacker needs access to an encryption oracle to see if the encryption of the plaintext guess matches the known ciphertext.

To defeat a chosen-plaintext attack, popular configurations of TLS use two common mechanisms: an initialization vector (IV) and a cipher block chaining mode (CBC). An IV is a random string that is XORed with the plaintext message prior to encryption — even if you encrypt the same message twice, the ciphertext will be different, because the messages were each encrypted with a different random IV. The IV is not secret; it just adds randomness to messages, and is sent along with the message in the clear. It would be cumbersome to use and track a new IV for every encryption block (AES operates on 16-byte blocks), so for longer messages CBC mode simply uses the previous ciphertext block as the IV for the following plaintext block.

The use of IVs and CBC is not perfect: a chosen-plaintext attack can occur if the attacker is able to predict the IV that will be used for encryption of a message under their control and the attacker knows the IV that was used for the relevant message, they are trying to guess

Thai Doung and Juliano Rizzo demonstrated that the above attack can be mounted against TLS under certain conditions. When an SSL 3.0 or TLS 1.0 session uses multiple packets, subsequent packets use an IV that is the last ciphertext block of the previous packet, essentially treating the session as one long message. This allows an attacker who can see encrypted messages sent by the victim to see the IV used for the session cookie, the because **cookie's location is predictable**, and also know the IV that will be used at the beginning of the next message packet (the last ciphertext block from the current message packet). If the attacker can also “choose” a plaintext message sent on behalf of the victim, they can make a guess at the session cookie and see if the ciphertext matches.
The ability of an attacker to mount a chosen plaintext attack against SSL/TLS with predictable IVs is well known. In essence the underlying cryptographic construct in SSL version 3.0 and TLS version 1.0 creates IVs by always using the last ciphertext block of packet number j as the IV for packet number j+1. What this means for an active attacker on a network is that they are able to sniff the IV for each record.

Suppose the attacker wants to guess that message mi may be secret x. She picks the next plaintext block to be $mj = mi \oplus c(i-1) \oplus c(j-1)$. Recall that the last block on ciphertext becomes the IV for the next block and that XORing two identical values cancels them out. cj then becomes:

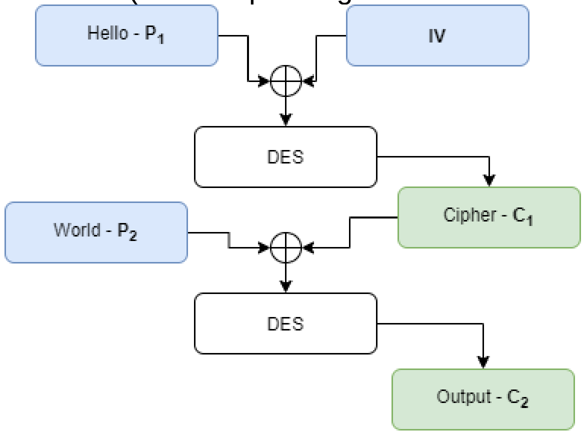
$$cj = E(k, c(j-1) \oplus mj) = E(k, c(j-1) \oplus c(j-1) \oplus mi \oplus c(i-1)) = E(k, mi \oplus c(i-1)).$$

If $c_j = c_i$ then the guess of secret x is correct. This attack, while interesting from a theoretical perspective, is not very practical. This is because an attacker can only guess whole blocks; assuming the secret x is a random value, then an attacker would have to guess on the order of the block size of the algorithm (e.g. 2^{128} for AES). However, if an attacker can split the secret over blocks under her control, she would be able to fixate on one byte of secret in a block that otherwise contains known information. The search space is therefore reduced to one-byte times the number of bytes to guess

Guessing the entire 16 bytes of a random session cookie is impractical, but the attack cleverly makes it easier by controlling the block boundaries (by adjusting URL parameters for example) in order to guess the secret one byte at a time. For example, say the actual plaintext session cookie is:
Company_Session_Token: Edx38NesewqeNI872Def32sfe
It would be difficult to guess 16-bytes of the unknown session cookie value if you have the ciphertext associated with the secret portion ``Edx38NesewqeNI87''. Instead, the attacker can try to adjust the encryption block boundaries and guess the plaintext value of a ciphertext block associated with an easier plaintext block: ` `_Session_Token:**?**"which only contains a single unknown character that is much easier to guess. After the first character has been correctly guessed as “E”, the attacker can guess the next character by shifting the block boundary to align with the plaintext block ``Session_Token:E?" and continue the attack until the entire cookie has been recovered. If the cookie is a random value that is base64 encoded, the attack would only take about 32 rounds per character in the cookie.

3 ON PAPER IMPLEMENTATION OF HOW AN ATTACKER WILL TAKE TO TRY TO EXPLOIT IT

Since TLS has a secure MAC along with it, so we as an attacker only have one block to compromise the communication. So, we are using CBC here and IV (initialisation vector) is random. Let's say the message transmitted is Hello World. First block would have hello and second block would have world (assume padding to make that the block size required for the



encryption algorithm)
So here we as an attacked don't know the second keyword but we know that the first keyword is hello and, in that way, we want to verify if the second keyword is World or not.

What the attacker can see by sniffing is C_1 and tries to perform packet injection of P_2 to verify if the output is the same, it would not work since it will xor with the CBC residue of the previous block which could be something totally different from C_1 let's call it CR.

But now we know if reapply xor with the same value it undoes the effect. So, we know that it would be xor-ed by CR. We would first xor it with CR and then C_1 and then make the sender send that packet. Now since the victim would xor this again with CR and the output is identical to C_2 we can verify that the second keyword was world

Injected Plaintext = ("World" $\oplus$ $C_1 \oplus$ CR) denoted by IP

If Encrypt ($P_2 \oplus C_1$) == C_2 == Encrypt(IP $\oplus$CR) then we can say P_2 is "World". Here the choice of encryption algorithm does not matter, in fact the victim could have used AES which is much stronger and we still would have been able to exploit the vulnerability.

To understand the BEAST approach to exploiting the known-CBC problem, recall that there are two problems, from the attacker's point of view, of record splitting attacks. The first is that the attacker has to "guess" an entire block of data to be able to discern any useful information. Suppose that instead of "world" there could have been a name and in that case, we would have to try out every name until we get lucky. The second problem is that the victim needs to be tricked into inserting known plaintext into the TLS stream in the first place.

Rizzo and Duong (authors of BEAST) solved this problem in a clever way, which is the novelty of the BEAST attack. It takes only one "hit" to determine if any given 8-byte block in the plaintext is precisely equal to the attacker's guess. If the attacker just wants to verify yes or no, then the record splitting attack is perfect. If, on the other hand, you want to determine the value of a block, you'd have to try every single one of the 256^8 combinations before you found the right one — on average, $256^8/2$. That's a lot of searching. However, if you already know the value of the first seven characters of the block, you only need to search through, on average, 128 combinations to find the value of the last one; fewer if you can employ some heuristics

So, how does one go about predicting these first 7 characters to take advantage of the design flaw? Well, one of the problems with HTTP from a security perspective is that it, like any useful network protocol, includes a lot of known plaintext for an attacker. Consider a fairly standard HTTP request. Parts of this request are required by the protocol to be in specific places, and many other parts are easily guessable, in bold the parts that are required and in italics the parts that take only a bit of guesswork to predict.

GET */index.html* **HTTP/1.1**
Host: *mysite.com*
Cookie: Session=123456
Accept-Encoding: text/html
Accept-Charset: utf-8

Here, the only really variable parts of the request are the page being requested (which in a lot of cases is itself easily guessable as well) and the value of the session cookie.

To sum it up we need some **prior** information about the location of keyword to make attack space manageable.

3.1 HOW TO SAFEGUARD AGAINST BEAST

One fix which was also proposed in TLS 1.1 was simple enough, theoretically - each packet gets its own IV, and that IV is transmitted (unencrypted) at the beginning of each packet. The fact that it's transmitted unencrypted is not a security problem because by the time the attacker can see it, it's already been used.

But sometimes, due primarily to client compatibility reasons neither TLS 1.1 or 1.2 are widely supported on the web and most vendors still require support (i.e. fall back) for SSL v3.0 and TLS v1.01. Browser vendors have attempted to implement a workaround to address the vulnerability at the implementation level while still remaining compatible with the SSL 3.0/TLS 1.0 protocol. These initially included inserting empty fragments into the message in order to randomize the IV as in the case of OpenSSL, and when that proved problematic to reliably implement, 1/n-1 record splitting where a single byte of the plaintext is injected in each record. The resulting padding added to complete the block (16 or 15 bytes) is random, and its search space is too high for an attacker to guess.

4 ABOUT CODE

Code simulates the process of how this attack would work. There needs to be a server and a client attached to each other with a proxy such that the man in the middle would be able to intercept messages, and would also have control over client side to send messages to the server.

To simplify the process, we receive a cookie which needs to be guessed and we have this prior information that it would always start with the "Cookie: " as discussed in earlier section we need this information. So, the simulation works in this way

1. Client sends first request with the cookie : Let's call this FR and we have a CBC residue let's call this Previous_IV
2. now we send another request along with it with the same message as the first request (note in this case IV would be Previous_IV) and observe the output, this would form our original output, and the last block would form the IV.
3. Now the idea is to guess the first block of the cookie therefore we make a guess with one of 256 characters appended to out prior. If out first block matches to the original block we have guessed correctly otherwise we would increment the i where ith character was guessed with until we find the missing character in the cookie
4. We keep repeating until we have found the cookie.

4.1 HOW TO RUN THE CODE

```
➜   python2 beast.py
Found character : V, cookie so far : V
Found character : e, cookie so far : Ve
Found character : R, cookie so far : VeR
Found character : y, cookie so far : VeRy
Found character : 5, cookie so far : VeRy5
Found character : e, cookie so far : VeRy5e
Found character : C, cookie so far : VeRy5eC
Found character : r, cookie so far : VeRy5eCr
Found character : e, cookie so far : VeRy5eCre
Found character : T, cookie so far : VeRy5eCreT
Found character : c, cookie so far : VeRy5eCreTc
Found character : o, cookie so far : VeRy5eCreTco
Found character : o, cookie so far : VeRy5eCreTcoo
Found character : k, cookie so far : VeRy5eCreTcook
Found character : i, cookie so far : VeRy5eCreTcooki
Found character : 9, cookie so far : VeRy5eCreTcooki9
Decoded cookie is: VeRy5eCreTcooki9
```

We can also try with a custom cookie instead of hardcoded one,

```
➜ python2 beast.py --cookie aNothEr5ecRetcOoKIE
```

5 REFERENCES

1. Attacks on SSL
2. Here Come The ⊕ Ninjas
3. An Illustrated Guide to the BEAST Attack
4. The Illustrated TLS Connection: Every Byte Explained
5. How the BEAST Attack Works