Team: Password Is Password

Mohit Jangid, Eric Lewantowicz, Joseph Shaffer

CSE 5473

4/22/2018

**Bleichenbacher and POODLE SSLv3 Padding Attacks**

**POODLE Attack**

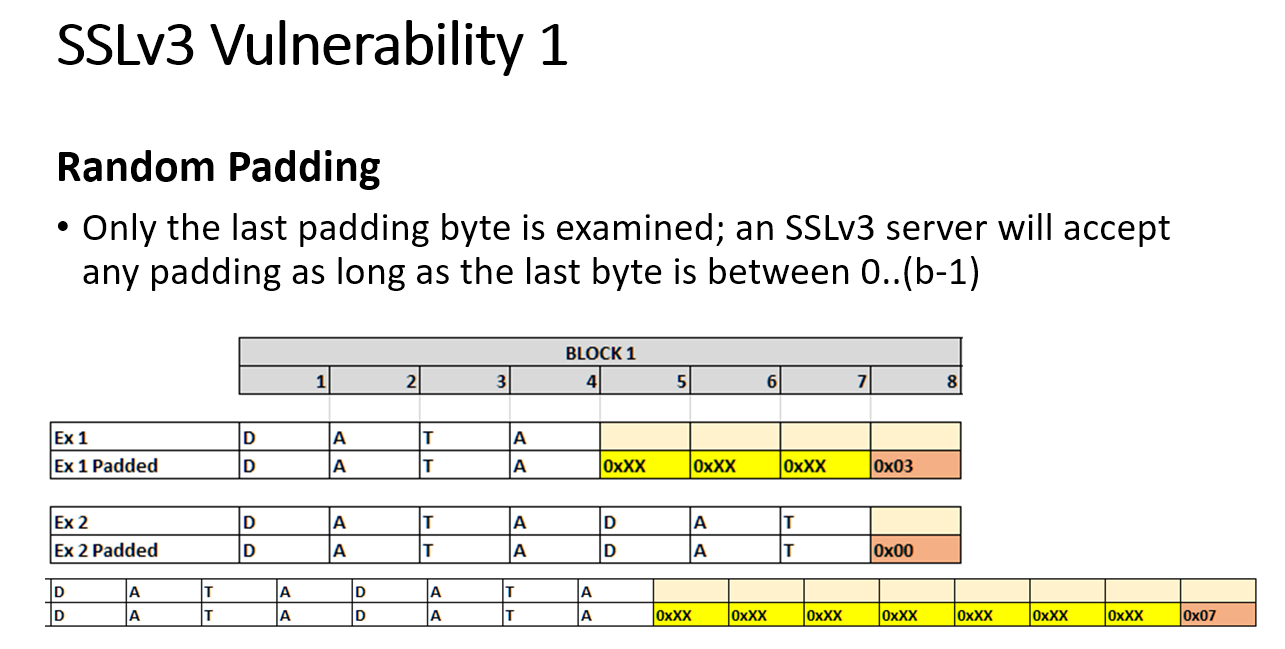
**Goals**

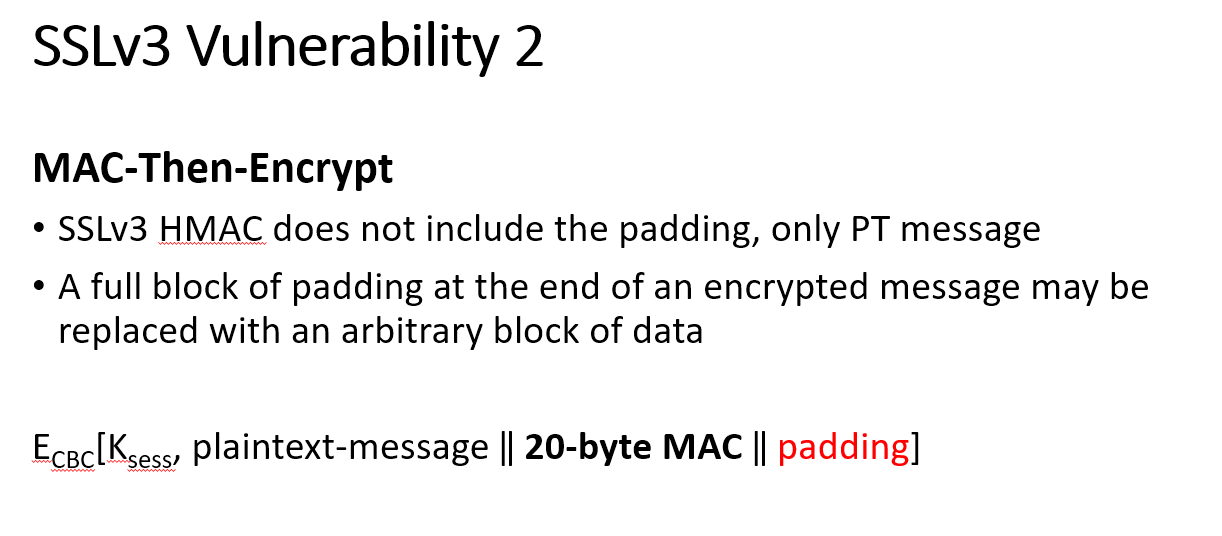
1. Learn the vulnerabilities of SSLv3 encryption scheme and the details of the POODLE attack.
2. Implement the SSLv3 protocol at a low-level from scratch in Python 2.7 with the Pycrypto module, with combined Client/Attacker communicating with Server Oracle over TCP stream sockets.
3. Write a GUI from scratch using Curses shell graphics module in order to provide an effective visualization of the attack simulation.
4. Consider possible defensive approaches and implement a defense against a POODLE attack.

The Padding Oracle On Downgraded Legacy Encryption attack was discovered by a Google Security team in 2014. It is a man-in-the-middle attack on the (at the time) deprecated but still supported SSLv3 encryption scheme using block CBC mode. Because many servers still supported SSLv3 in order to support legacy applications and browsers still employing it, an attacker could execute a “downgrade dance” to cause a browser and server to fallback to an SSLv3 encryption scheme. Using elements from the BEAST attack (see references), an attacker can inject an HTTPS request generator into a victim client’s browser using JavaScript injection from a complicit HTTP website. The attacker could then use the request generator to fashion plaintext messages, that combined with a secret cookie, would be encrypted with SSLv3 and then sent to an HTTPS server. The attacker intercepts the encrypted traffic, moves the ciphertext block containing the secret cookie to the end of the message, replacing a full block of padding, and then forwards the modified encrypted message to the server. The server acts as an oracle and disconnects the session if the padding or MAC is incorrect, and accepts the message if the padding is correct. This process allows the attacker to decrypt the contents of the message, i.e. the cookie, byte by byte.

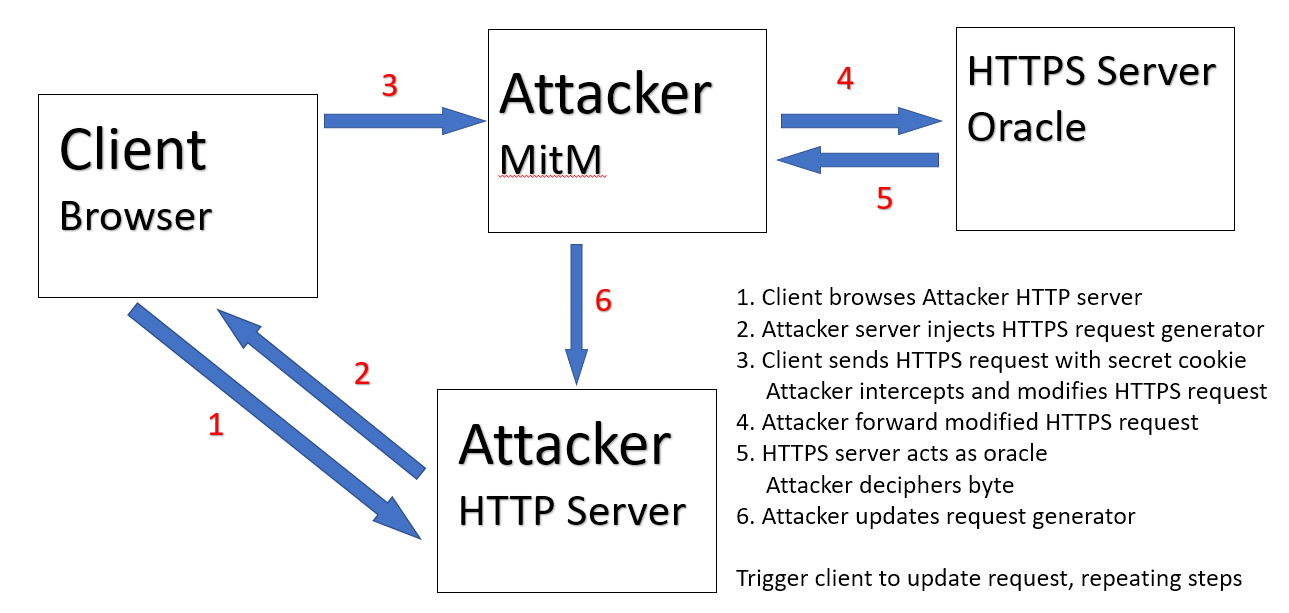
After discovery of the POODLE attack, the long-term solution was to eliminate SSLv3 support altogether in all browsers and servers, since the RC4 stream variation of SSLv3 has its own vulnerabilities and is not considered secure. However, several short-term mitigations were proposed and implemented in the short term, including anti-POODLE record splitting by the Opera browser. While we didn’t find any details on their record-splitting approach, the idea served as the seed for our own record splitting defense proposal.

**Key SSLv3 vulnerabilities that the POODLE attack exploits** (graphics from our PowerPoint presentation)





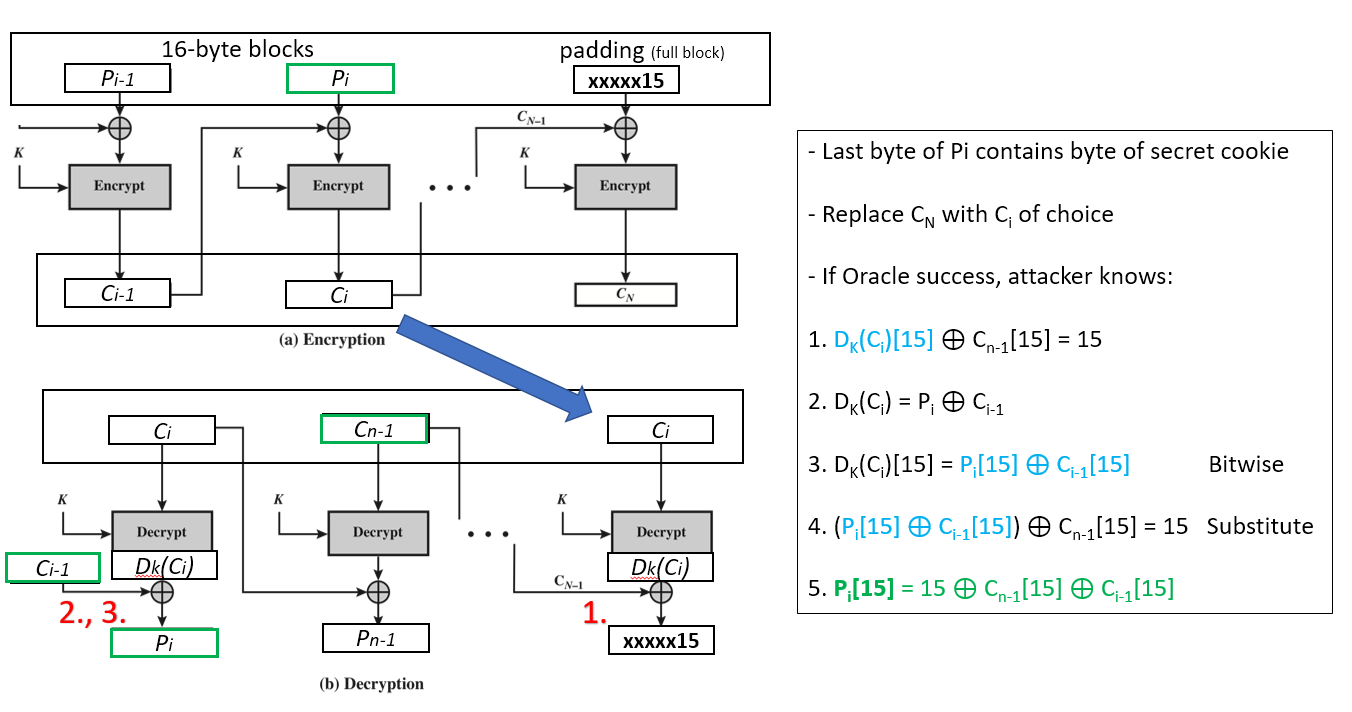
**Attack overview** (diagram from our PowerPoint presentation)



**Key Attack Steps**

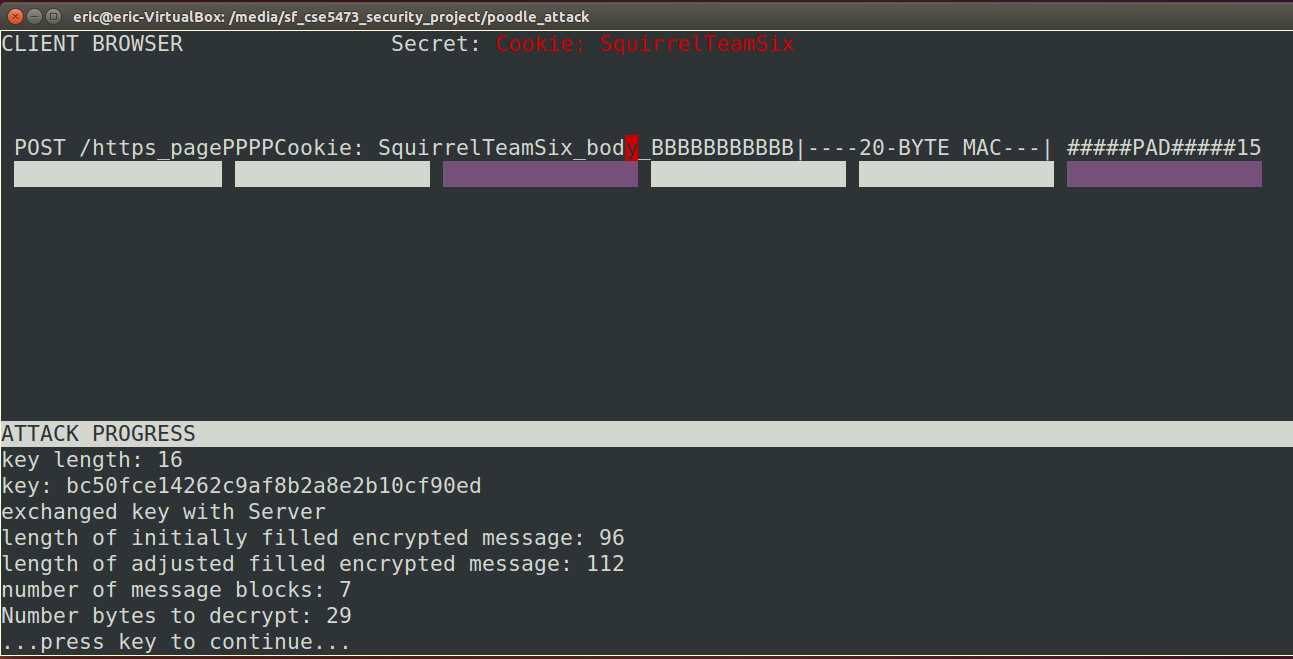
1. Add initial fill bytes to the HTTPS request path and body.
2. Intercept modified encrypted request, C0||C1||...Cn, and continue to add fill bytes to the HTTPS request body until the modified encrypted message increases by a block length. The IV is appended to the encrypted message as C0 from the client.
3. Key concept: now the attacker knows a full block of padding exists in block Cn, with no MAC contamination. This block can be replaced by a block Ci without affecting the MAC.
4. Replace block Cn with block Ci and forward the request to the server oracle.
5. If the server accepts the block with correct padding/correct MAC, the last byte of block Ci can be decrypted (see details in graphic below)
6. If the server rejects the block with incorrect padding/MAC, the attacker triggers the client to re-encrypt the same request. A new IV is used, so the encrypted contents are changed, Cn is again replaced with Ci, and the message is again sent to the server. This process is repeated until the server accepts the block.
7. After decrypting the last byte of Ci, attacker request generator removes one fill byte from the request body, and inserts an additional fill byte to the path. This shifts the cookie/data blocks one byte to the right while maintaining a full block of padding.
8. Repeat steps 4-7 until all of the cookie bytes are decrypted.

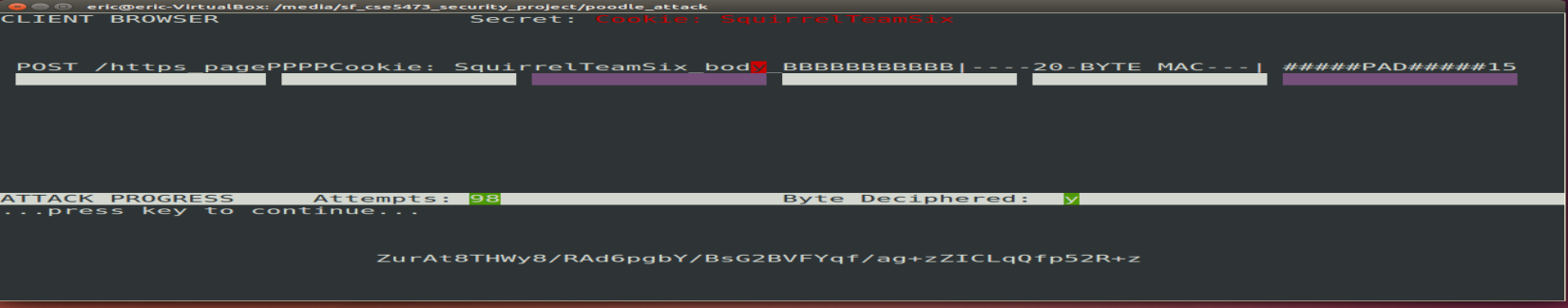
**Low-level attack process visualized** ((graphic from our PowerPoint presentation

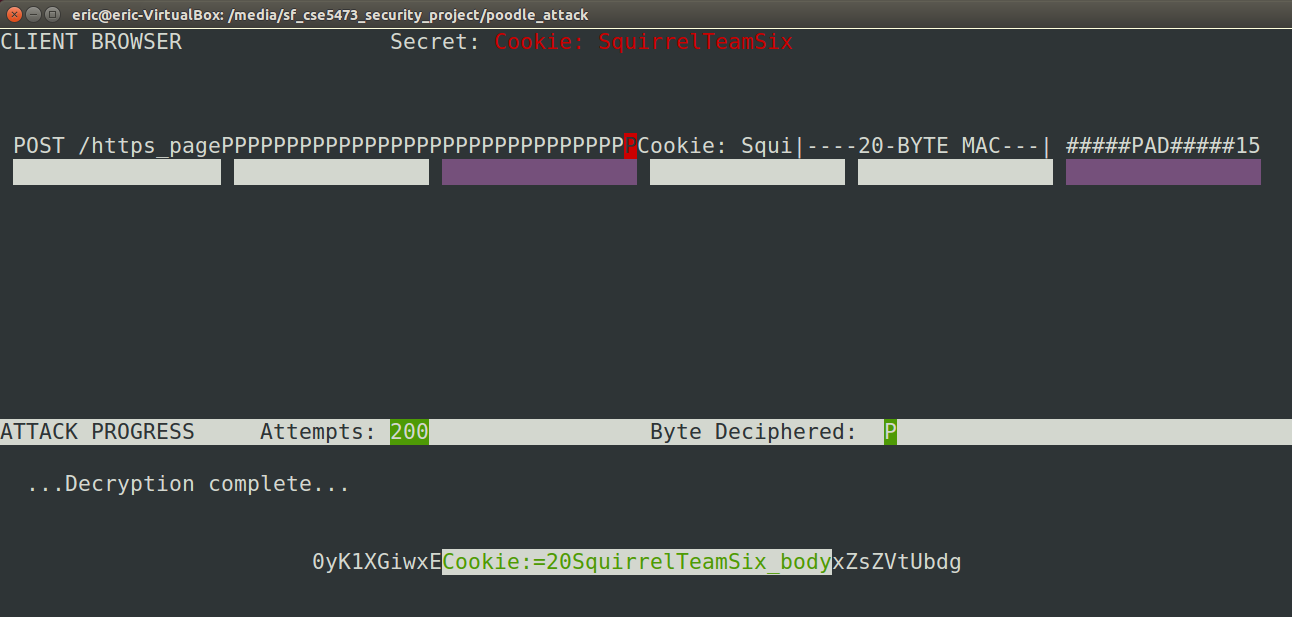


**Our Implementation and Visualization of the Attack and Defense using Curses shell graphics module**







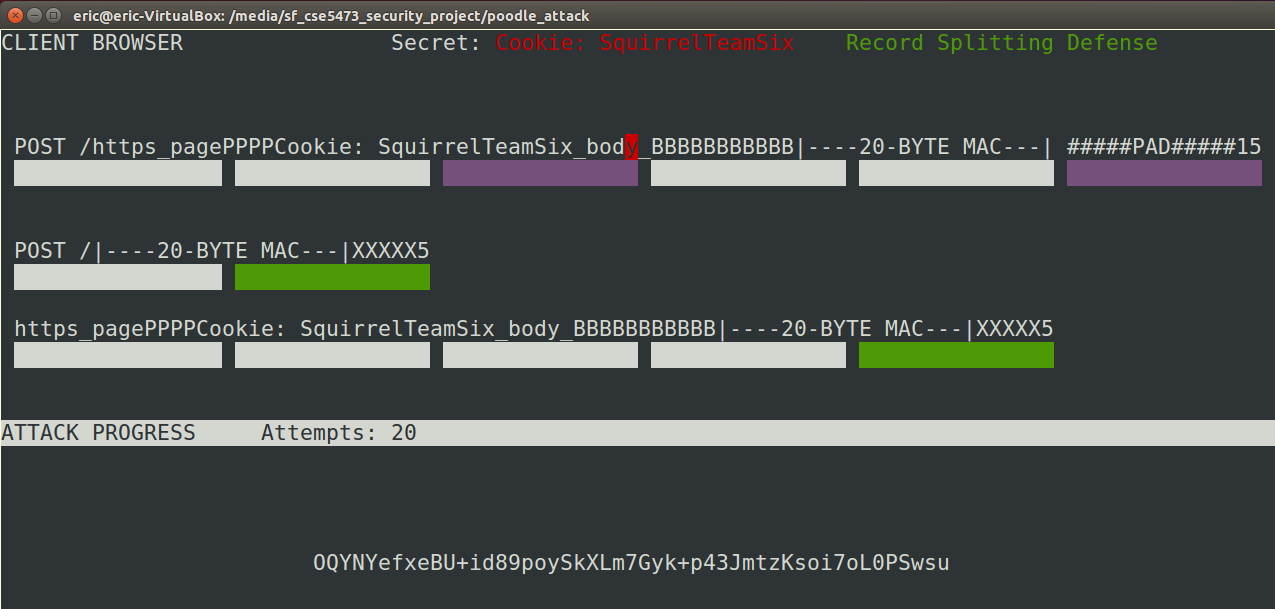


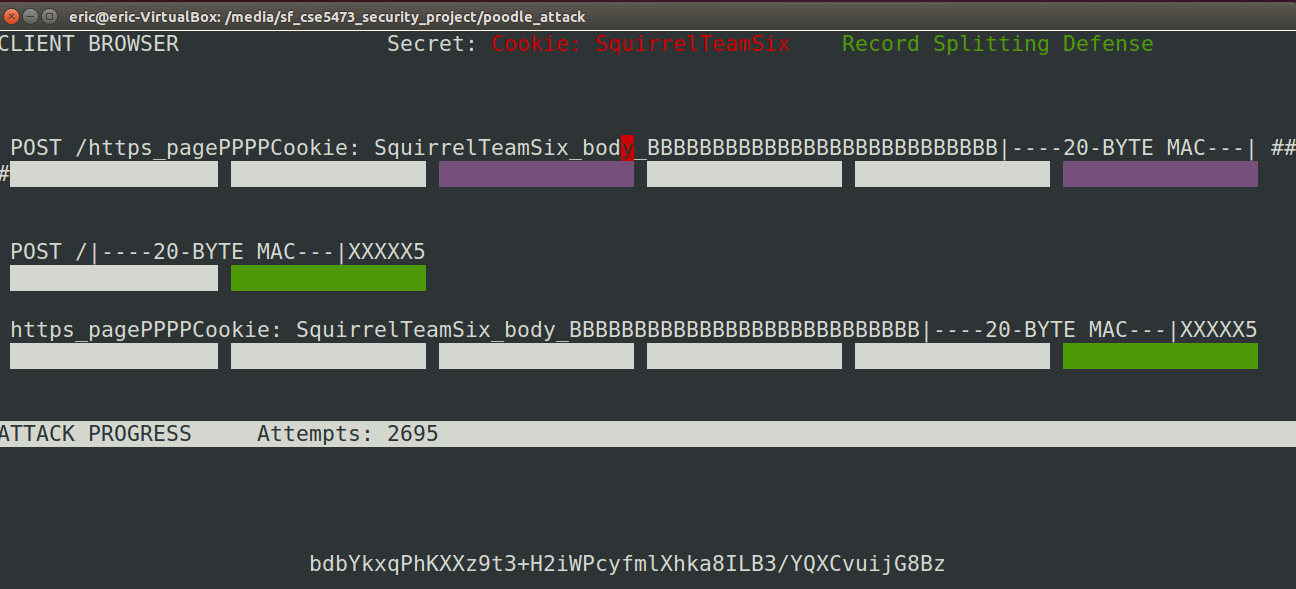
**Defense Proposal & Experiment**

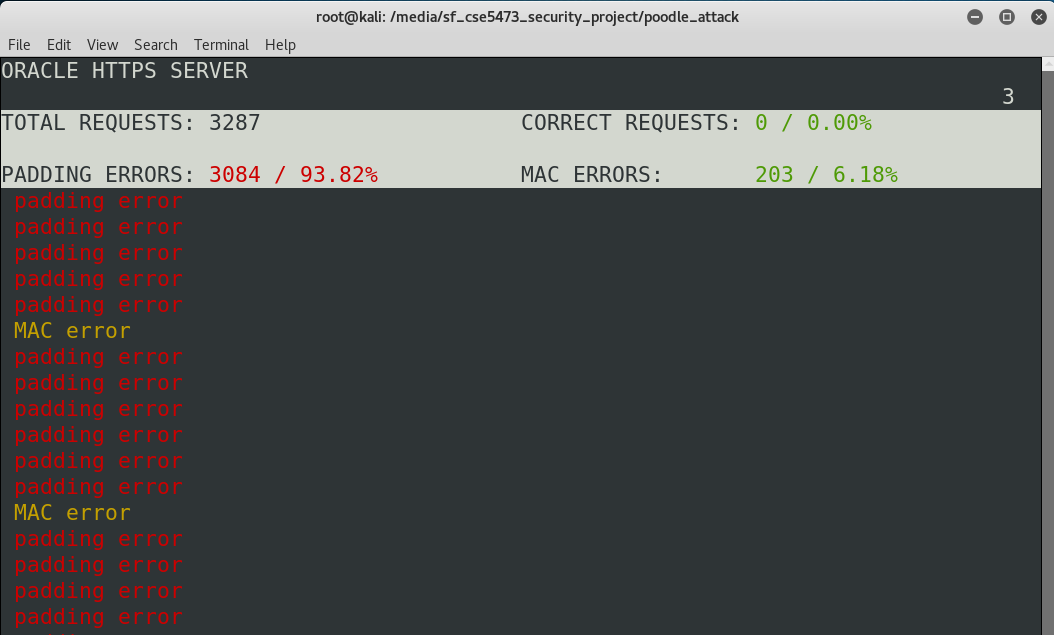
The attack is a very low-effort attack, with a successful response from the server occurring every 1/256 attempts, until an IV / Key combination produce a final padding byte in Ci that is decrypted to a value of 15 (x0F). The Opera browser implemented some temporary anti-POODLE record splitting whose details we didn’t find, but gave us the idea to split vulnerable records before encryption, in order to ensure that a full block of padding would never exist, and that sufficient bytes of the MAC would be pushed into the final block Cn, in order to sufficiently decrease the odds of a successful oracle response.

A 6 / n-6 record splitting approach seems to make intuitive sense, as it pushes 10 bytes of MAC into the final blocks of both split records. A record split would be executed at the browser above the encryption layer anytime that it receives a message length where length(message +MAC) % 16 ==0, i.e. a message that would result in a full block of padding. Upon identifying this condition, the browser splits the record into two records. The first record contains the first 6 bytes of the plaintext message, and the second record contains the remaining n-6 bytes. Both records are then independently MAC’d, padded, and encrypted. The effect is that the 20-byte MAC is pushed into the final message block, so that both messages contain 10 bytes of MAC, and 6 bytes of padding. This now decreases the probability that an attacker can replace the final block and achieve a successful match of both the final byte of padding AND 10 bytes of MAC in the last block from 1/256 to 1/288 = 3.23x10-27. This makes a successful attack much less likely. As seen below, the attacker is not successful after 3,000 oracle requests. Of course the ultimate solution is to disable SSLv3 altogether, as modern browsers have now done, but this record splitting may server as an effective interim solution. This defense approach could be further modified to split records with fewer than X MAC bytes in the final block, say if an attacker attempted to submit records with 15 bytes of padding and a single byte of MAC in the final block in an attempt to avoid a record split, if the cost of a 1/216 oracle success was still an affordable cost of effort.

One of the drawbacks to this defense is that it requires servers to correctly handle assembly of split records upon receipt, which has a non-trivial likelihood of causing interoperability problems between browsers employing a record-splitting defense and servers.







**Bleichenbacher Attack Overview**

The Bleichenbacher attack is an adaptive chosen ciphertext attack against certain protocols based on RSA that was introduced by Daniel Bleichenbacher in 1998. Bleichenbacher shows that an RSA private-key operation can be performed if the attacker has access to an oracle that, for any chosen ciphertext, returns only one bit telling whether the ciphertext corresponds to some unknown block of data encrypted using PKCS #1. SSL V.3.0 is a protocol that is susceptible to this attack.

The attacker uses the oracle to decrypt or sign a message. To do so, the attacker sends ciphertexts to the oracle and by combining the returns from the oracle, the attacker can gain information on cd. The chosen ciphertexts sent to the oracle are based on the previous outcomes of the oracle, which is why the attack is called an adaptive chosen ciphertext attack. This attack is considered to be very practical, as it is easy to get necessary information based on the replies from the oracle. The attack can be carried out if the attacker, for example, has access to a server that accepts encrypted messages and returns an error message depending on whether the decrypted message is PKCS conforming.

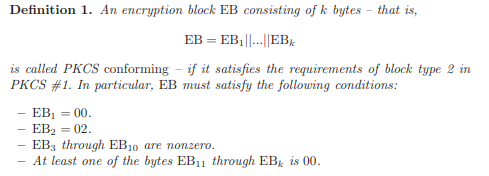
**PKCS #1**

To better understand the attack, the RSA encryption standard PKCS #1 must be described. The standard consists of three block formats, block types 0 and 1, which are reserved for digital signatures, and block type 2 which is used for encryption. Block type 2 is the main block format that is relevant to the attack.



In RSA, n, e are the public key and p, q, d are the secret key. n = pq and d ≡ e−1 (mod ϕ(n)). Furthermore k represents the byte length of n. Hence, we have 28(k−1) ≤ n < 28k. A data block D consists of |D| bytes and is encrypted as follows. The first step is that a padding string PS, which consists of k-3-|D| nonzero bytes is generated pseudo-randomly. The byte length of PS is at least 8, because |D| must not exceed k-11. Thus, the encryption block EB = 00||02||PS||00||D is formed and is converted into an integer x and is encrypted with RSA. This means that the ciphertext c ≡ xe (mod n).

An important aspect to understand is how the receiver parses the ciphertext. By decrypting the ciphertext with the private key, the receiver gets an integer x’. Then x’ is converted into an encryption EB’. From there, the receiver looks for the first zero byte, which indicates the ending of the padding string PS and the start of data block D.



A ciphertext c is PKCS conforming if its decryption is PKCS conforming.

**Chosen-Ciphertext Attacks and Bleichenbacher Attack**

In a basic chosen-ciphertext attack, the attacker selects the ciphertext and sends it to the victim. In return the corresponding plaintext or some part thereof is given. The Bleichenbacher attack is called adaptive because unlike a basic chosen-ciphertext attack, the attacker can choose the ciphertexts depending on the previous outcomes of the attack.

Plain RSA is susceptible to a chosen-ciphertext attack because the attacker can choose a random integer s and ask for the decryption of the message c’ = sec mod n. From the answer m’ = (c’)d and it is easy to recover the original message due to the fact m = m’s-1 (mod n).

Furthermore, another aspect of RSA encryption is that the least significant bit is as secure as the whole message. This means that if there exists an algorithm that can predict the least significant bit of a message given only the corresponding ciphertext and public key, then there is an algorithm that can decrypt a ciphertext. Thus an attacker does not need to learn the complete decrypted message in a chosen-ciphertext attack, but instead can rely on decrypting single bits per chosen ciphertext.

For the Bleichenbacher attack, it is assumed that the attacker has access to an oracle, which returns whether the corresponding plaintext is PKCS conforming for every ciphertext. The Bleichenbacher attack’s goal is to minimize the number of chosen ciphertexts and wants to take advantage of the specific properties of PKCS #1. The main aspect that the attack relies on is that the first 2 bytes of the PKCS #1 format are constant and that these two bytes are known with certainty when a ciphertext is accepted.

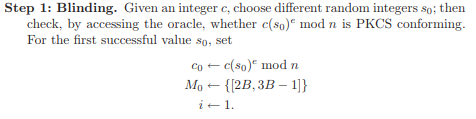
**Description of Bleichenbacher Attack**

The attacker wants to find m = cd (mod n), where c is an arbitrary integer. Basically, the attacker chooses integers s and computes c’ = cse (mod n) and sends c’ to the oracle. The oracle will then determine if c’ is PKCS conforming and if the oracle says that c’ is PKCS conforming, then the attacker knows that the first two bytes of ms are 00 and 02. Furthermore, we know B = 28(k-2) and k is the length of n in bytes. This means that if ms is PKCS conforming, then we know 2B ≤ ms mod n < 3B.

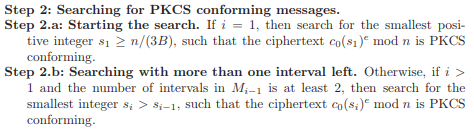
m can eventually be derived by collecting several such pieces of information. About 220 chosen ciphertexts will be sufficient to derive m.

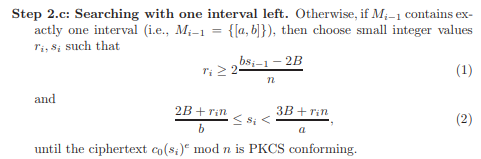
Bleichenbacher describes the attack in 3 phases, where the variable Mi will always be a set of closed intervals that is computed after a successful si has been found, such that m0 is contained in one of the intervals of Mi.

The first phase consists of the message being blinded, where a ciphertext c0 that corresponds to an unknown message m0 is given.

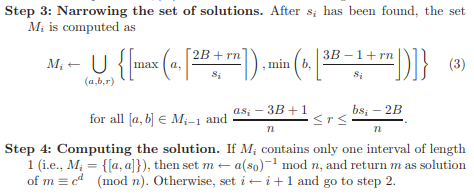


For the second phase, the attacker will try to find small values si, for which the ciphertext c0(si)e mod n is PKCS conforming. For each successful value of si, the attacker will compute a set of intervals that must contain m0 using previous knowledge about m0.

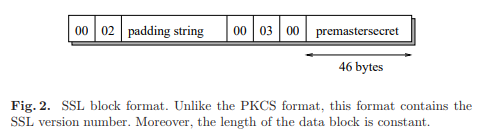




When one interval remains, the third phase begins. The attacker at this point has sufficient information about m0 to choose si, such that c0(si)e mod n is much more likely to be PKCS conforming than is a randomly chosen message. The size of si is increased gradually, which narrows the possible range of m0 until only one possible value remains.



**SSL V.3.0**



SSL V.3.0 is susceptible to the Bleichenbacher attack during the SSL V.3.0’s handshake protocol. When the client and server first exchange the client.hello and server.hello messages, which among other information exchanges, the cryptographic routines are selected. After that, the client and server may send their public key and certificates. The client will then generate a random secret bit string, which is called the pre\_master\_secret, which is encryted with RSA and the resulting ciphertext is sent to the server, which decrypts the ciphertext. If the server finds that the resulting plaintext from the decrypted ciphertext is not PKCS conforming, the server will send an alert message to the client and will close the connection. Otherwise, if the resulting plaintext is PKCS conforming, the server will continue the handshake protocol. The client then will send a finished message, which contains strong authentication and the client will need to know the pre\_master\_secret to compute the message.

An attacker cannot complete the handshake protocol successfully without being able to generate a finished message based on the pre\_master\_secret. However, the attacker does not have to complete the handshake protocol because the attacker gets the necessary information, whether the chosen message is PKCS conforming before the protocol is finished. Thus, by constantly checking if the chosen messages are PKCS conforming, the attacker gradually learns more and is able to determine the original message of the client.

**Bleichenbacher Attack Example**

This example is a simplified version of the Bleichenbacher attack.

In this example:

N = 41

2B = 10

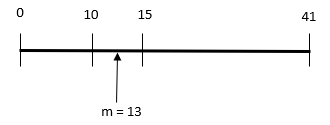
3B = 15

We know that 2B = 00 || 02 || (00)k-2 and 3B = 00 || 03 || (00)k-2 and thus any message that lies between 2B and 3B is PKCS conforming, because if you recall the PKCS format (k bytes) is 00 || 02 || padding string || 00 || original message.

Thus if m is PKCS conforming, then 2B < m < 3B.

And if ms mod N is also PKCS conforming then 2B < ms mod N < 3B

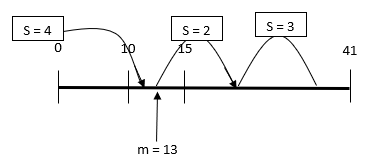
For this example, m =13, but our goal is to figure that out and we do not know that yet.



To start, the attacker has intercepted the ciphertext c = xe mod n, where x is the message m converted to an integer x. The attacker will take the ciphertext c and will choose an integer s to calculate c’ = csmod n = m\*se mod n and send c’ to the oracle to determine if c’ is PKCS conforming.

First the attacker will choose s = 2 and find c’ = 2\*m = 26 as m = 13 and send c’ to the oracle, which will be determined not to be PKCS conforming. The attacker will then choose s = 3 and c’ = 39 and finally choose s = 4, and c’ = 52 mod 41 = 11 to find c’ is PKCS conforming.

Furthermore, originally you know that m has to lie between 2B = 10 and 3B = 15.



Thus, now you know that 4\*m mod n is PKCS conforming and falls between 2B = 10 and 3B = 15.

If you extend the number line from 0 to 2n, you know that any message that is PKCS conforming will fall between 51 and 56, because 10 + 41 = 51 and 15 + 41 = 56.

After you find s = 4, where s \* m mod n = 4 \* 14 mod 41 = 11 is PKCS conforming, you know that ms mod n, where s = 4 falls between 10 and 15, and you know m falls between 10 and 15. However, now you can narrow down that interval to help determine the true value of m using the s value found.

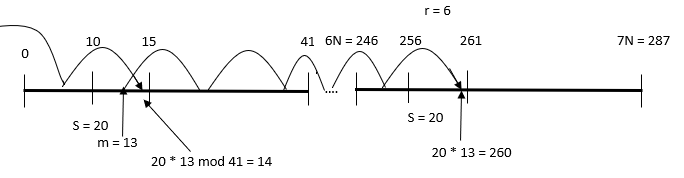
The attacker calculates the new interval for m using 2B <= m\*s – N < 3B, and from this we can conclude that (2B + N) / s <= m < (3B + N) / s for the new interval. This means that the new interval is (10 + 41) / 4 <= m < (15 + 41) / 4, which equals 12.75 <= m < 14. Furthermore, this means the interval is about 13 <= m < 14 because the message is converted to an integer, which means m cannot be 12.75 and since m has to be an integer that is greater than or equal to 12.75, m has to be at least 13. This means that m falls in the range of 13 and 14 instead of 10 and 15.

The attacker will repeat this process of finding a value of s where m \* s mod n is between the interval of 13 and 14. The next s that is found is 20, where 20 \* m mod n = 20 \* 13 mod 41 = 14, which does fall in the range of 13 and 14.

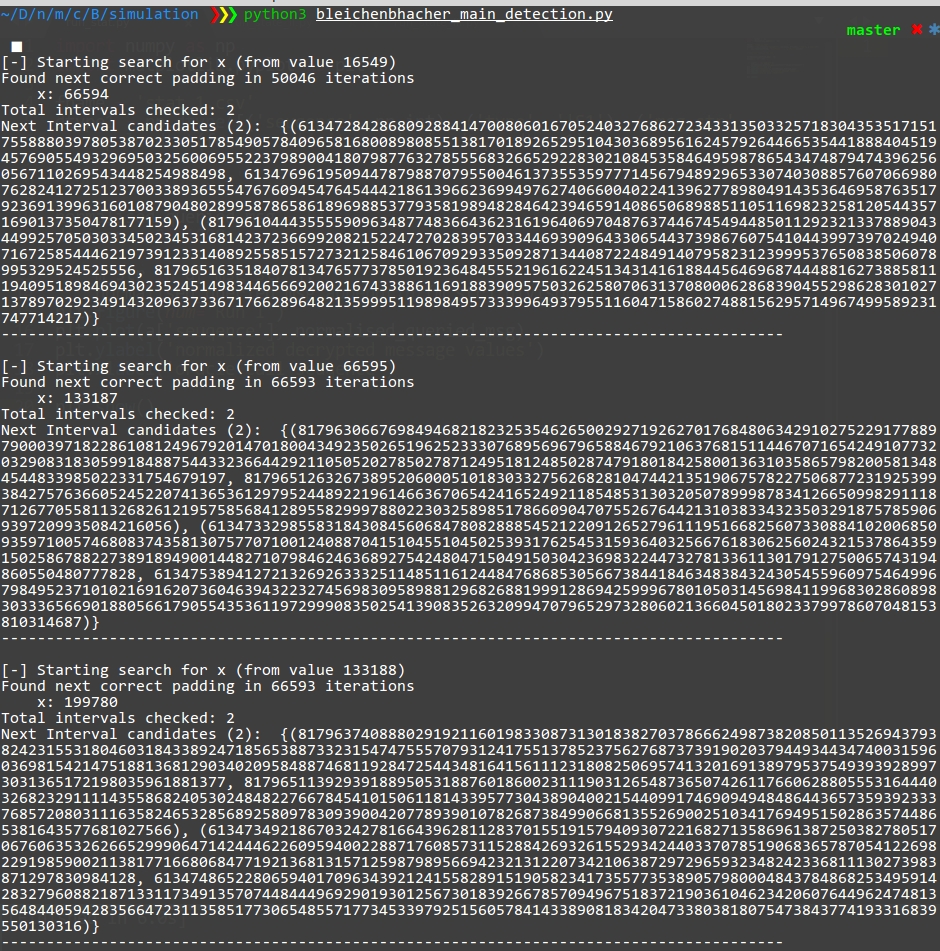
The next step is to extend the number line to help determine the new interval. We know that 20 \* 13 mod 41 = 14, and thus 20 \*13 falls between 2B + r \* N (10 + r \* 41) and 3B + r \* N (15 + r \* 41) for some value of r. In this case, r = 6 and 20 \* 13 = 260, which falls between 10 + 6 \* 41 = 256 and 15 + 6 \* 41 = 261.

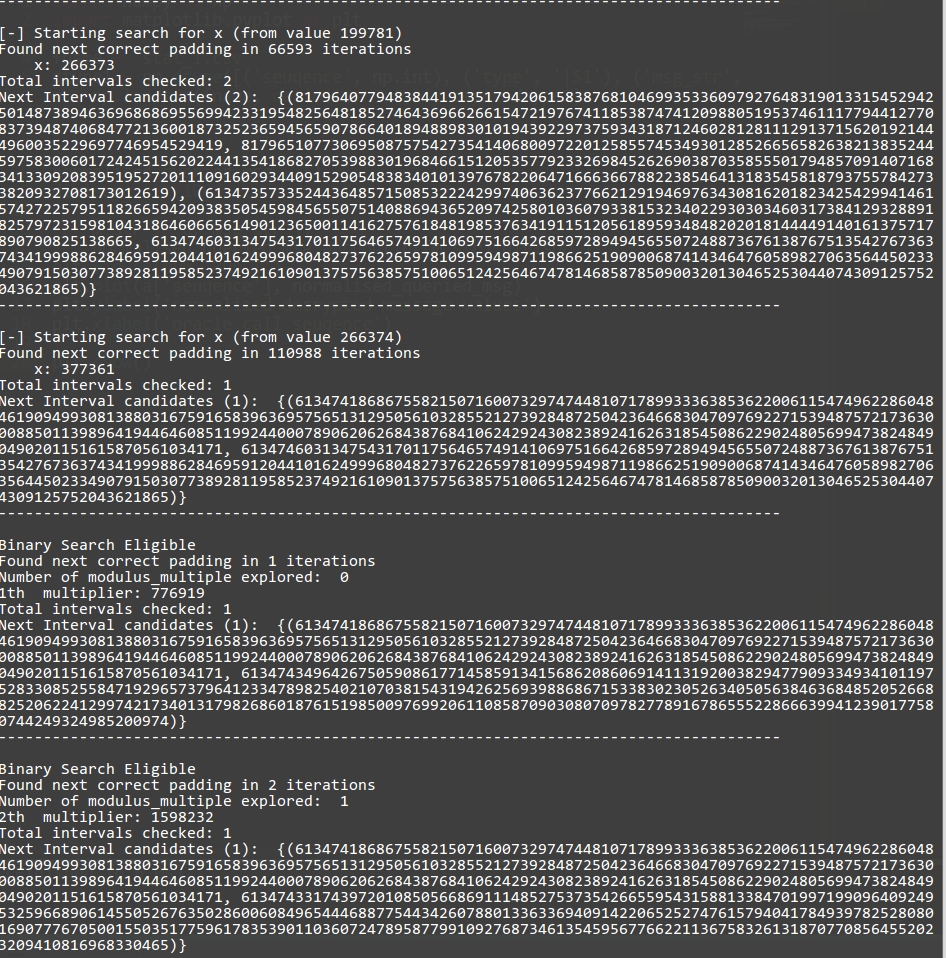
Now we can calculate our new interval with (2B + rN) / s <= m < (3B + rN) / s. Thus the new interval is 256/20 = 12.8, which we can rounded up to be 13, since m is an integer that has to be greater than or equal to 12.8 and 261/20 =13.05, which can be rounded down to 13 because m has to be an integer less than 13.05. Thus, from there the attacker can determine that m = 13 and the attacker has recovered the original message.

The left side of the figure below shows that 20 \* 13 mod 41 = 14 and wraps around the number line, while the right side shows the extended number line where 20 \* 13 = 260 and r = 6 so 2B + r \* N = 10 + 6 \* 41 = 256 and 3B + r \* N = 15 + 6 \* N = 261.

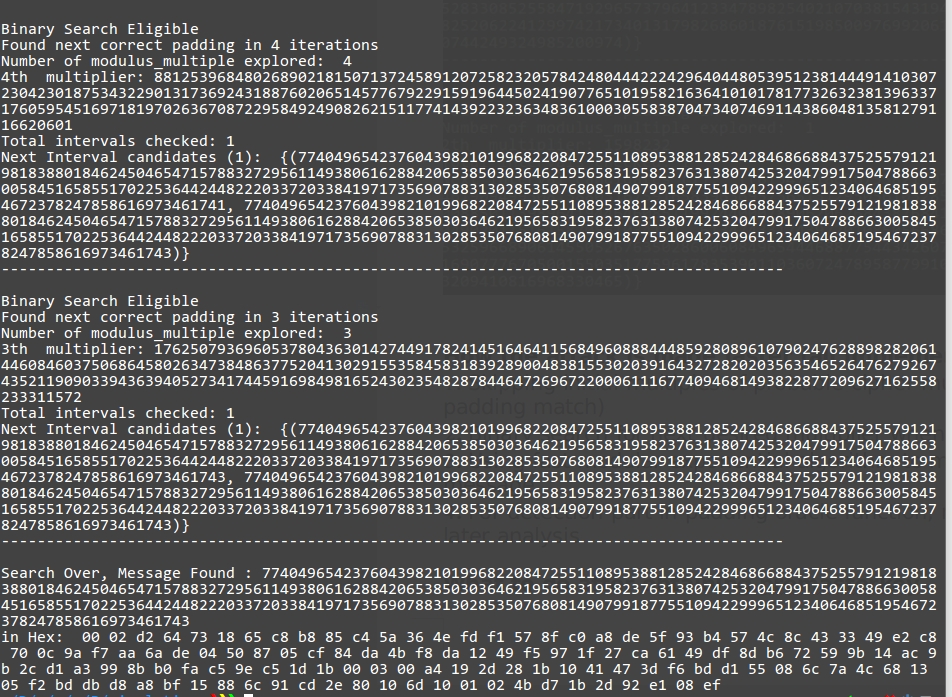


**Demo Run of Simulation code (1024 bit RSA keys are used)**

****

****

**~~ skipping to last part~~**

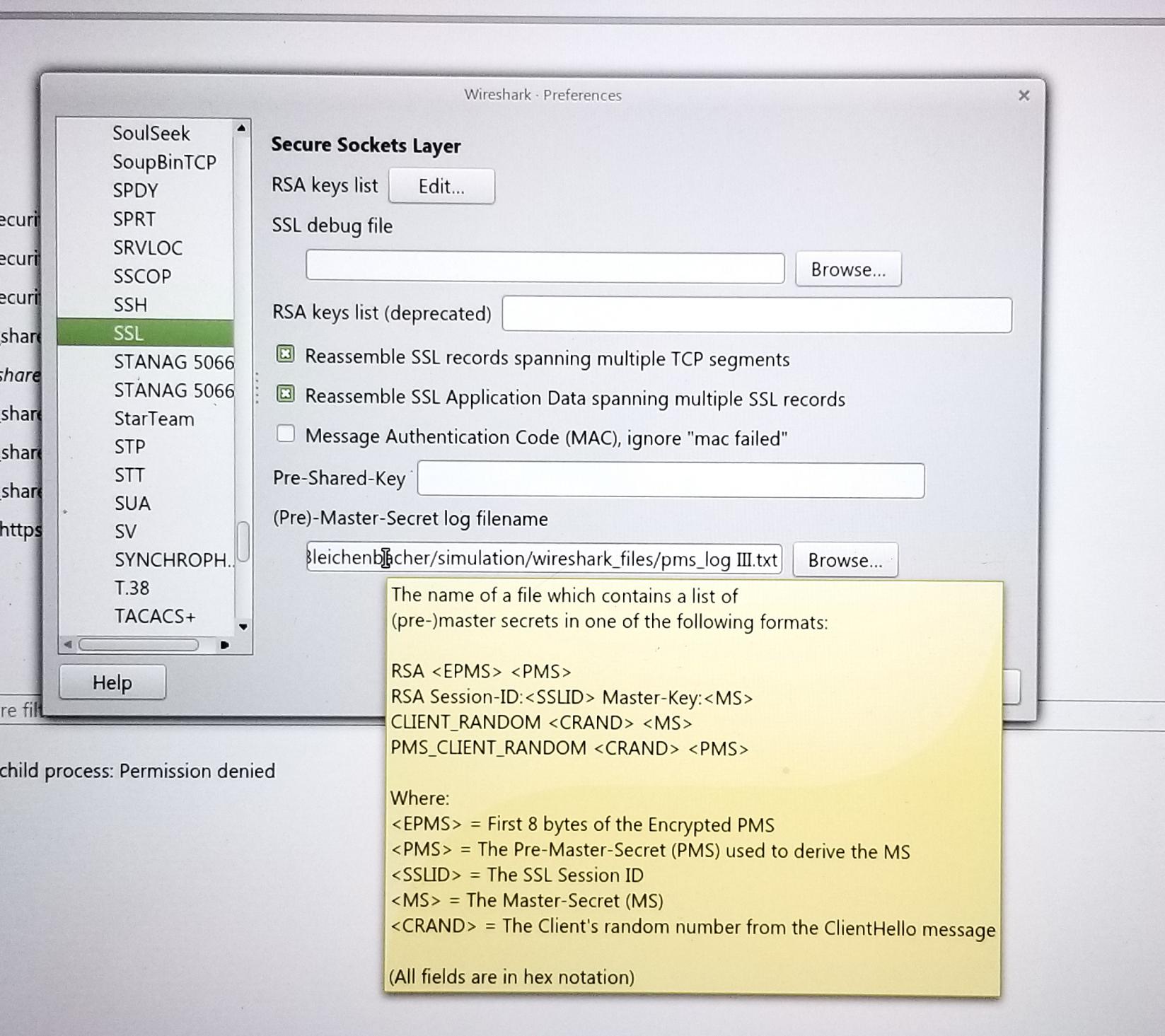
****

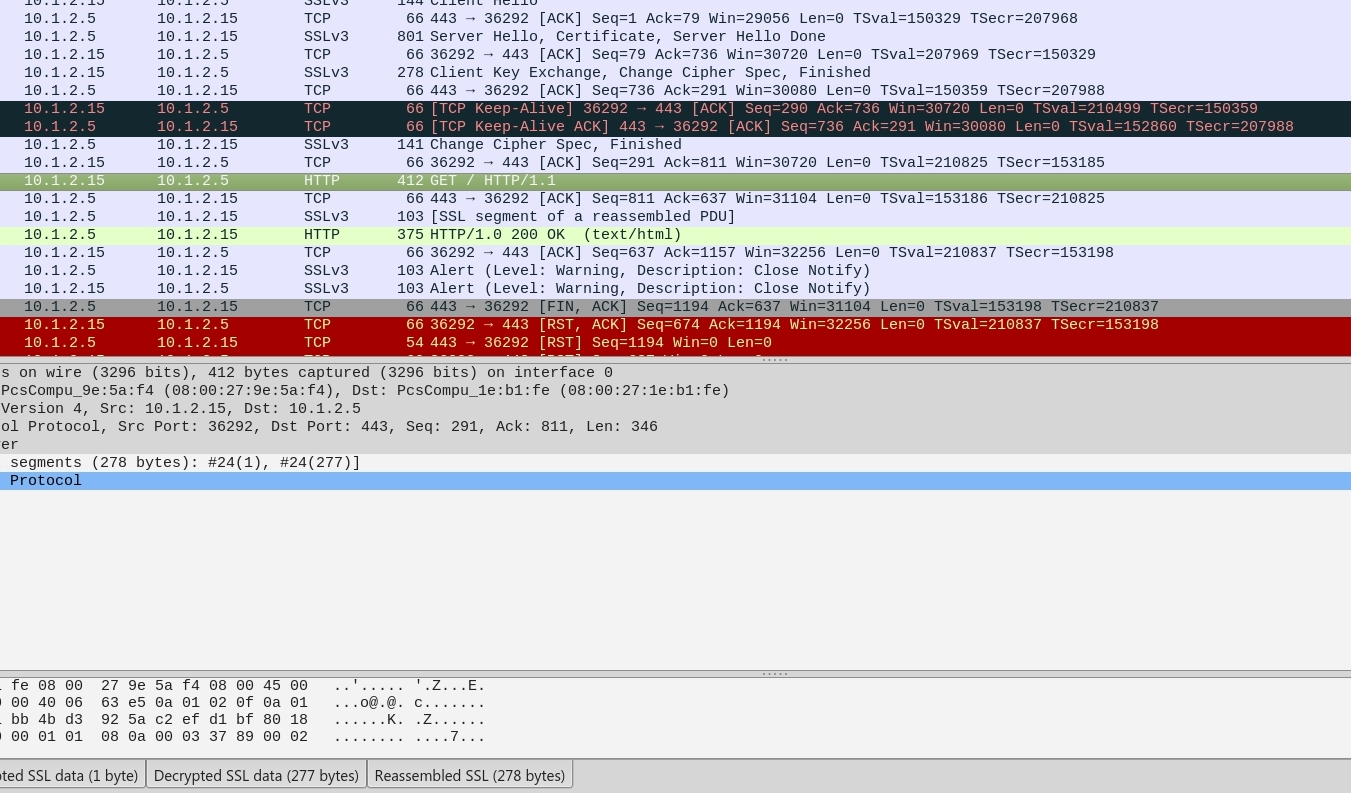
**How the attack is made faster (all implemented in the simulation)**

1. Skipping initial multiples of probable cipher multiplier (valid for only first search up to correct padding match)
2. Binary search for next desired multiplier when only one possible shrinked interval is left.
3. Using fast modular exponential algorithm to multiply x^e to the cipher and hence x to the message
4. For detection part in padding oracle function, minimal detection statistical data are stored for later analysis

**What to do after obtaining premasterkey**

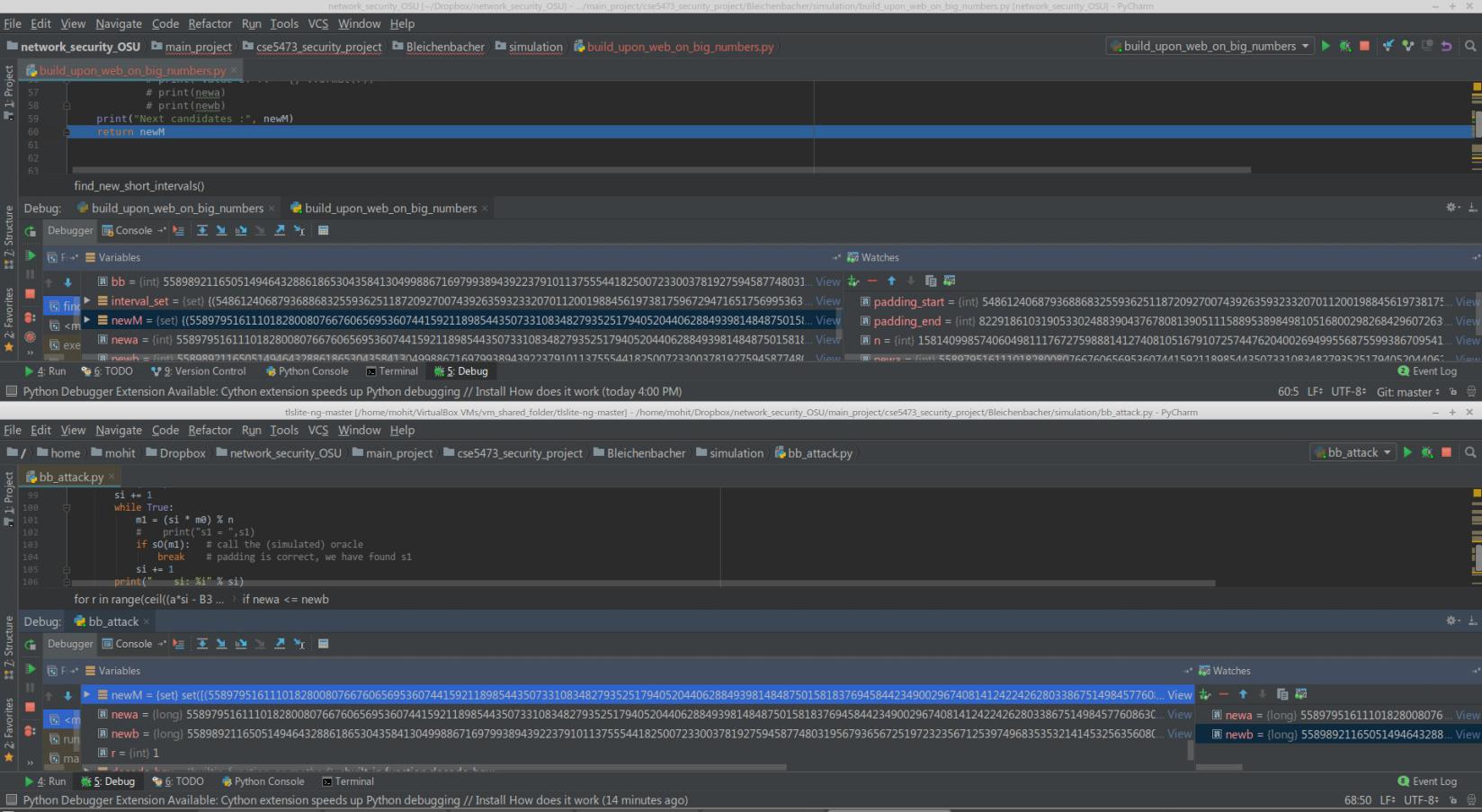
Create a PMS\_log\_file with the desired format according to Wireshark. Pass the log file to Wireshark for SSL settings and plain HTML text for SSLV3 encrypted data can be seen: -

****

****

**Challenge Faced in Bleichenbacher Attack project work**

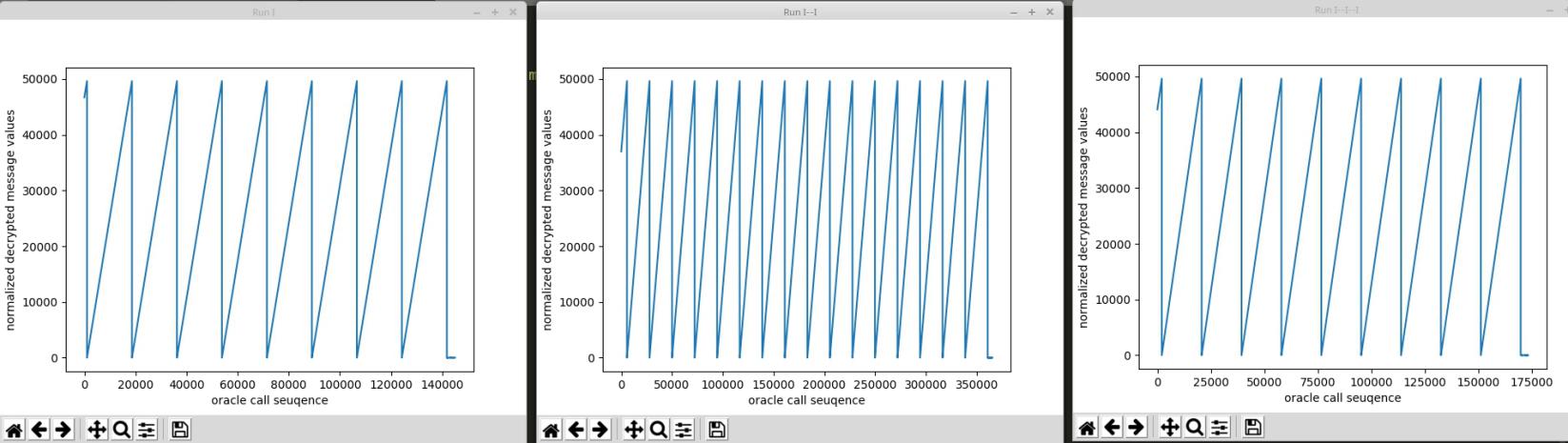
1. Conceptual understanding of attack and improvements. Especially how the mathematical construct are working behind the scene.
2. Coding with big numbers, debugging, python2 and python3 differences. Following is screenshot of debugging big numbers on attack code in pycharm.

****

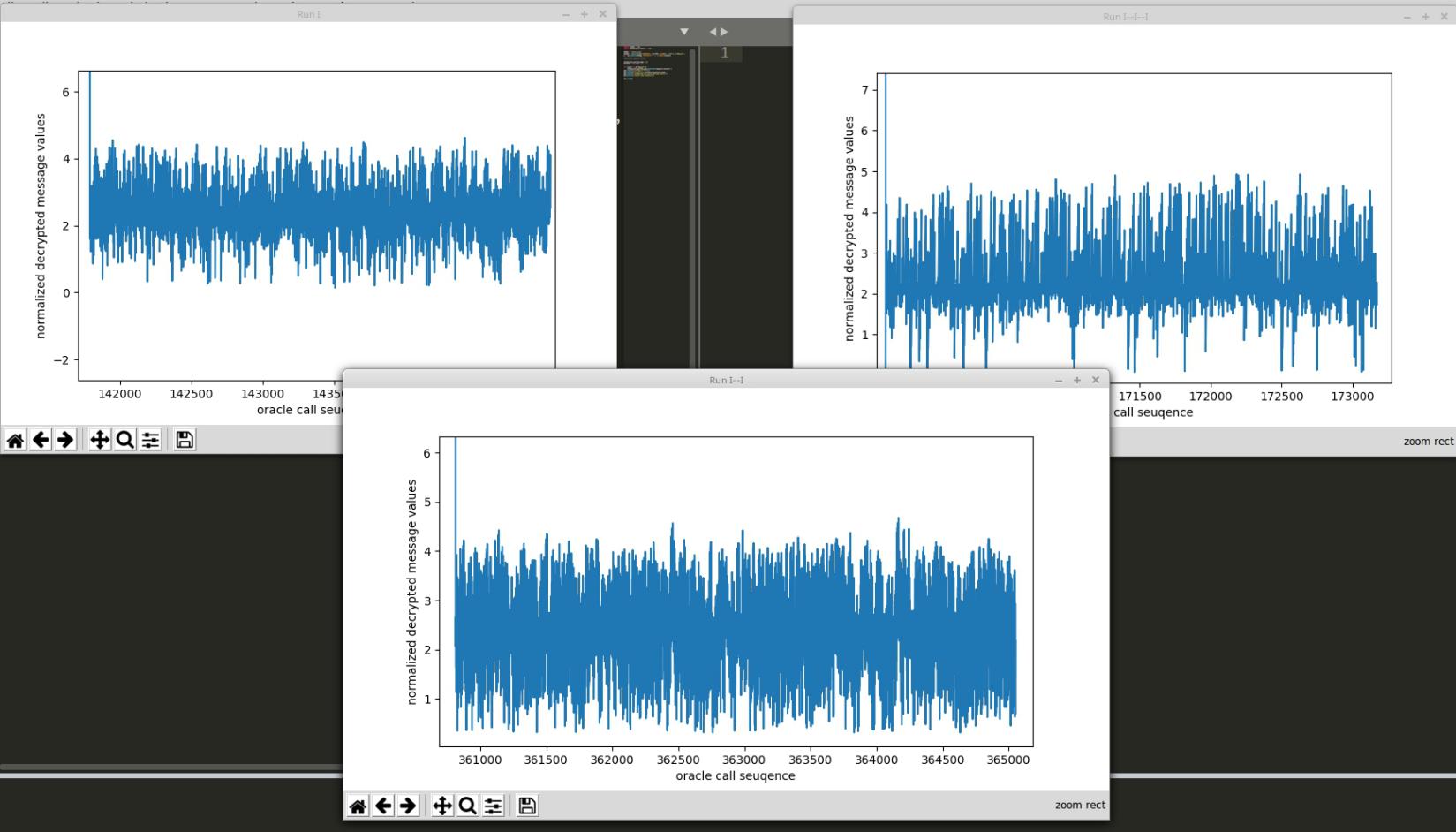
1. Last but not the least, the limited time for project work given

**BleichenBacher Attack Detection**

The main idea behind the detection is to identify the oracle queried messages. In the padding oracle we stored the decrypted messages in to a file for later analysis. All the decrypted message are converted in to integer and are normalized by dividing all messages integers by 2^1008. The initial line plot for three run of the attack on different wireshark capture of encrypted premasterkey data are following



The x axis just a count value for all received cipher to padding oracle. The y axis represent normalized decrypted message inter values. The repeated pattern is obvious as modular operation is applied on message integers. The initial small starting curve reaching up to top represent skipping of initial multiples of probable cipher multiplier (valid for only first search up to correct padding match). Thereafter the multiplier is searched linearly and hence the repeated triangles. Interestingly the last flat tail which seem trivial on these images represent binary search improvement. However, the flat tail is actually flat because the x and y axis values are still so huge. After zooming in to last flat tail here is what we observe:



As we can see the even binary search pattern for 3 difference instance of same attacks on different ciphers gives us some characteristic curves. These characteristics curves combined with whole graph plot can help us detect the attack.

**References**

This POODLE Bites: Exploiting The SSL 3.0 Fallback, https://www.openssl.org/~bodo/ssl-poodle.pdf

POODLE attacks on SSLv3 (14 Oct 2014), Imperial Violet blog, https://www.imperialviolet.org/2014/10/14/poodle.html

[BEAST] T. Duong, J. Rizzo: “Here Come The ⊕ Ninjas”, 2011.

Bleichenbacher, Daniel (1998). ["Chosen Ciphertext Attacks Against Protocols Based on the RSA Encryption Standard PKCS #1"](https://web.archive.org/web/20120204040056/http:/www.bell-labs.com/user/bleichen/papers/pkcs.ps). CRYPTO '98: 1–12. Archived from [the original](http://www.bell-labs.com/user/bleichen/papers/pkcs.ps) (PS) on 2012-02-04. Retrieved 2011-12-07

Practical Padding Oracle Attacks on RSA, Riccardo Focardi, http://secgroup.dais.unive.it/wp-content/uploads/2012/11/Practical-Padding-Oracle-Attacks-on-RSA.html