CS 742 Foundations of Network Security and Cryptography
Assignment 3

Ajay Kedare	Astha Jada	Neha Garg	Sankalp Rangare
153059007	153050027	153050039	153050087

October 30, 2015

1 Re-design the Needham-Schroeder protocol to use as few messages as possible. Prove informally that your protocol is secure. If you could use timestamps in lieu of (or in addition to nonces), what is the smallest number of messages you need to achieve the same functionality provided by the Needham-Schroeder protocol? Show your new design.

1. $A \rightarrow KDC : M, A, B, R1$
2. $KDC \rightarrow A : M, \{T_s, V, K_{ab}, B, R1, \{T_s, V, K_{ab}, A\}K_{bs}\}K_{as}$
3. $A \rightarrow B : \{T_s, V, K_{ab}, A\}K_{bs}, \{T_a\}K_{ab}$
4. $B \rightarrow A : \{T_a + 1, Tb\}_{K_{ab}}$
5. $A \rightarrow B : \{T_b + 1\}_{K_{ab}}$

Here, T_a , T_b and T_s are the timestamps.
M is session identifier.
V is validity of message.

$A \rightarrow KDC : M, A, B, R1$
First, A sends to KDC, message identifier, and nonce R1 along with "A" and "B".

$KDC \rightarrow A : M, \{T_s, V, K_{ab}, B, R1, \{T_s, V, K_{ab}, A\}K_{bs}\}K_{as}$
Now the key part is when KDC sends the timestamp and validity to A and also includes it in the ticket for B which is encrypted with K_{bs}. Ticket, $\{T_s, V, K_{ab}, A\}K_{bs}$ contains the actual session key between A and B i.e. K_{ab}.

$A \rightarrow B : \{T_s, V, K_{ab}, A\}K_{bs}, \{T_a\}K_{ab}$
A verifies the nonce R1 and forwards the ticket recieved from KDC to B along with new timestamp T_a encryted with session key K_{ab}.

$B \rightarrow A : \{T_a + 1, Tb\}_{K_{ab}}$
B decrypts the ticket, and collect the session key K_{ab} and also verify that the timetamp from server and validity mentioned is correct or not. If it is not then B can reject this message. B again sends his timestamp to A and verifies the timestamp challenge of A

$A \rightarrow B : \{T_b + 1\}_{K_{ab}}$
A now decrypts the message with the session key K_{ab} and sends verification for B's timestamp.

Hence any replay attack is not possible since B is verifying the the timestamp of B's ticket generation and its validity. Also while sending the message to A, B is veryfying the A's timestamp and sending back the B's timestamp to be verified by A.

2 Study the following attacks on SSL or on OpenSSL – BEAST, Heartbleed, Fluke. For each attack answer the following: What does the attack accomplish? Desribe a "typical" attack scenario. What is the vulnerability behind the attack? How can the attack be defended against?

2.1 Beast

2.1.1 What does the attack accomplish?

BEAST Attack which stands for *Browser Exploit Against SSL/TLS Attack* is based on "chosen plain text attack". TLS 1.0 uses CBC (Cipher Block Chaining) mode of operation for encryption. CBC modes makes use of previous cipher text block C_{i-1} as the initialization vector (IV) for encrypting the next plain text block P_i.

If an attacker is sniffing the encrypted data sent by the browser, then he/she will be able to obtain the IV that will be used for the next message encyption. So if the attacker can make a guess at the session cookie (P_i) and see if this new cipher text (C_i') matches with original cipher text (C_i) then attacker has made a correct guess at the session cookie.

2.1.2 Desribe a "typical" attack scenario.

Considering that attacker has identified a block of ciphertext Ci that he/she wants to decrypt, becuase it contains the session cookie. Now as he already sinffed the network so he already has Ci-1 with him. As per CBC mode of operation: $C_i = Ek(P_i \oplus C_{i-1})$ Attacker will write a Javascript program to transmit any chosen plain text block P_i that he/she wants to encrypt. Then it will produce a ciphertext C_i' that attacker can sniff. Suppose attacker has correctly guessed the plain text P_i then attacker will generate $P_i' = IV \oplus P_i \oplus C_{i-1}$. Now with the help of the javascript which attacker has created he will force P_i to be replaced with P_i'. Now TLS layer will encrypt it as:

$C_i' = Ek(IV \oplus P_i')$
After putting P_i' value we'll get
$C_i' = Ek(IV \oplus IV \oplus P_i \oplus C_{i-1})$
IV $\oplus$ IV will cancel out each other we will get
$C_i' = Ek(P_i \oplus C_{i-1})$
If this C_i' is identical to C_i we can conclude that attacker has correctly guess the plain text block P_i. If they are not identical then attacker needs to guess the plain text block untill he/she finds a match.

2.1.3 What is the vulnerability behind the attack?

As TLS 1.0 uses CBC (Cipher Block Chaining) mode of operation for encryption. CBC modes makes use of previous cipher text block C_{i-1} as the initialization vector (IV) for encrypting the next plain text block P_i.

2.1.4 How can the attack be defended against?

This attack can be avoided by instead of directly using previous cipher text block as IV we can have explicit IV for each block to be encypted. This attack was addressed in TLS 1.1 and TLS 1.2 by the use of "explicit IVs" for each block.

2.2 Heartbleed

OpenSSL versions 1.0.1 through 1.0.1f (inclusive) are vulnerable to the heartbleed attack. In SSL when client communicates to the server it regularly sends heartbeat messages to the server to see if the server

is alive. If the server is alive it sends reply back to the client. Both server and client sends out messages regularly to make sure both the parties are alive. This feature is used in the heartbleed attack.

The heartbeat message consists of a payload, typically a text string, along with the payload's length as a 16-bit integer. The receiving computer then must send exactly the same payload back to the sender. The vulnerable versions of OpenSSL allocate a memory buffer for the message to be returned based on the length field in the requesting message, without checking the actual size of that message's payload. As proper bound checking is not done the message returned consists of the payload, possibly followed by whatever else happened to be in the allocated memory buffer.

2.2.1 What does the attack accomplish?

Heartbleed is therefore exploited by sending a malformed heartbeat request with a small payload and large length field to the vulnerable party permitting attackers to read up to 64 kilobytes of the victim's memory.Attackers in this way could receive sensitive data, compromising the confidentiality of the victim's communications like username,passwords etc. The worse thing is that sometimes the private key can also be exposed and the attacker which would enable attackers to decrypt communications.

2.2.2 Desribe a "typical" attack scenario.

A Heartbeat Request might ask a party to "send back the three-letter word 'abc'", resulting in a response of "abc" while a malicious heartbeat request of "send back the 500-letter word 'abc'" would cause the victim to return "abc" followed by whatever 497 characters the victim happened to have in active memory.As described above this may cause the victim to send sensitive data to the attacker.

2.2.3 What is the vulnerability behind the attack?

The vulnerability of this attack lies in the OpenSSL implementation. Here the receiver of heartbeat message does not check the bounds i.e if the length is same as the message received.

2.2.4 How can the attack be defended against?

The attack can be defended by doing proper bound checking. Also if the private key is compromised it should be revoked and new private key should be used, else the attacker may decrypt the message or can also change the message.

2.3 Freak

FREAK (Factoring RSA Export Keys) is a security exploit of a cryptographic weakness in the SSL/TLS protocols.The vulnerability was introduced during 1990's, when the US government banned selling strong cryptographic software overseas, unless it used export cipher suites which involved encryption keys no longer than 512-bits so it can be decrypted by NSA.

The need to support export cipher suites led to some technical challenges. Since U.S. servers needed to support both strong and weak cryptographic algorithms, the SSL designers used a 'cipher suite' negotiation mechanism to identify the best cipher both parties could support. As a result the strong cipher can be used when available and weaker otherwise.

2.3.1 What does the attack accomplish?

Using this attack an attacker can impersonate as the client and exchange messages with the server. The message can also be modified by the attacker.

2.3.2 Desribe a "typical" attack scenario.

FREAK attack basically is a Man in the Middle(MITM) attack that uses export cipher suite. The attack can be done as follows:

1. The client asks the server for the standard 'RSA' ciphersuite.
2. The MITM attacker changes this message and asks for 'export RSA'.
3. The server then responds with a 512-bit export RSA key, signed with its long-term key.
4. The client accepts this weak key due to the bug in OpenSSL implementation.
5. The attacker factors the RSA modulus to recover the corresponding RSA decryption key.
6. When the client encrypts the 'pre-master secret' to the server, the attacker can now decrypt it to recover the 'master secret'.
7. From here on, the attacker sees plaintext and can inject anything it wants.

2.3.3 What is the vulnerability behind the attack?

The vulnerability behind this attack is that export cipher suites are still supported by some of the servers and they can be easily decrypted.

2.3.4 How can the attack be defended against?

The server should disable support for any export suites. Administrators should be encouraged to disable all insecure ciphers and enable suites which support perfect forward secrecy i.e if a session key is revealed it should not affect the future messages and they cannot be decrypted.

3 Buffer overflow vulnerability

3.1 On linux machine(x86_64)

We completed buffer overflow attack using x=56 and y=7.

1. We disabled canary variable and make Stack executable and compiled program using:
`gcc -g -fno-stack-protector -z execstack -o prog prog.c`

2. Now we started to debug the program using **gdb**.
We added a break point in function **A( )** and analysed the program.
We checked the address of local varibales of function **A( )**.

```
(gdb) break A
Breakpoint 1 at 0x400537: file 1.c, line 7.
(gdb) r
Starting program: /home/sankalp/crypto3/prog

Breakpoint 1, A (m=5, n=3) at 1.c:7
7       int z = m + n;
(gdb) s
10      ptr = buf2 + 56;
(gdb) p &buf2
$1 = (char (*)[10]) 0x7fffffffded0
(gdb) p &buf1
$2 = (char (*)[5]) 0x7fffffffdee0
(gdb) p &ptr
$3 = (char **) 0x7fffffffdef0
(gdb) p &z
$4 = (int *) 0x7fffffffdefc
```

Figure 1: Address of local varibales.

After getting the address of local variables we 'disassemble' the main function and checked the return address of function **A( )**.

```
(gdb) disass main
Dump of assembler code for function main:
   0x0000000000400562 <+0>:     push   %rbp
   0x0000000000400563 <+1>:     mov    %rsp,%rbp
   0x0000000000400566 <+4>:     sub    $0x10,%rsp
   0x000000000040056a <+8>:     movl   $0x1,-0x4(%rbp)
   0x0000000000400571 <+15>:    mov    $0x3,%esi
   0x0000000000400576 <+20>:    mov    $0x5,%edi
   0x000000000040057b <+25>:    callq  0x40052d <A>
   0x0000000000400580 <+30>:    movl   $0x2,-0x4(%rbp)
   0x0000000000400587 <+37>:    mov    -0x4(%rbp),%eax
   0x000000000040058a <+40>:    mov    %eax,%esi
   0x000000000040058c <+42>:    mov    $0x400624,%edi
   0x0000000000400591 <+47>:    mov    $0x0,%eax
   0x0000000000400596 <+52>:    callq  0x400410 <printf@plt>
   0x000000000040059b <+57>:    leaveq
   0x000000000040059c <+58>:    retq
End of assembler dump.
```

Figure 2: Return address of function A()

3.Stack layout:

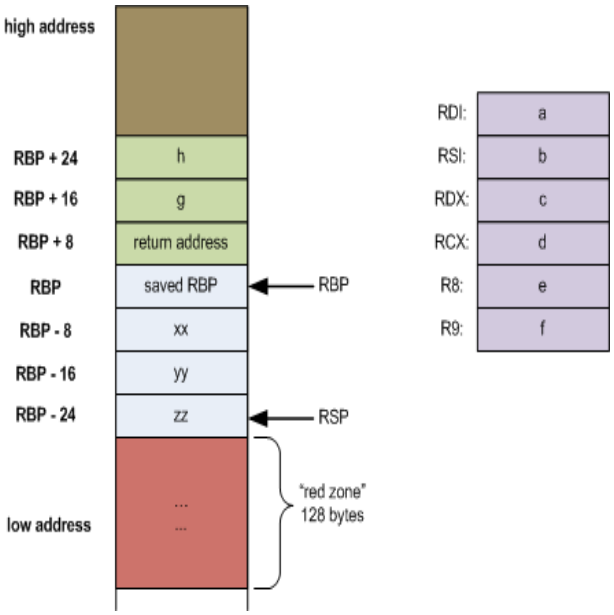

Figure 3: Stack frame layout

As we can see in stack frame return address is stotred in (base-pointer+8) address.
We check the content of registers and get the address of base-pointer stored in register **rbp**.

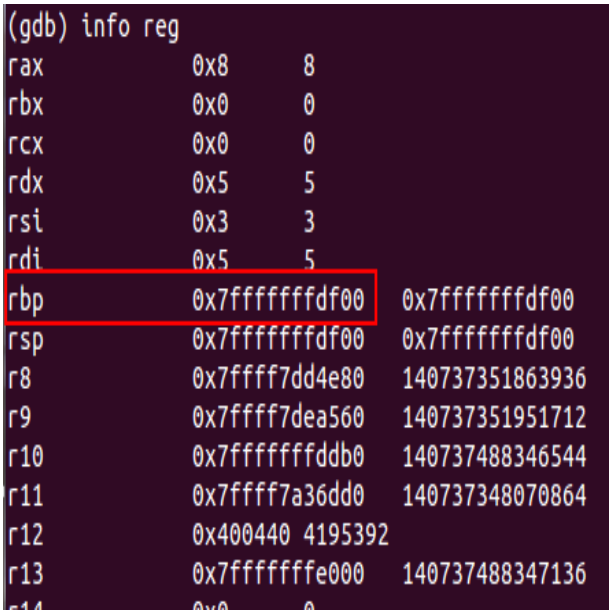

Figure 4: Base-pointer address

(base-pointer address + 8) will give the location where return address is stored.

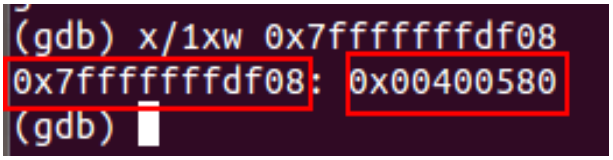

Figure 5: Location of return address in stack-frame

Now,we have the location of **return address** and address of buf2.
We calculated the differnce between these two addresses,which will give value of **x=56.**
As we can see from the assembly code of main function,next address after the return address of **A()** is **0x400587**.
So we change the return address by adding **7** to the content of ***ptr** variable.
Therefore value of **y=7.**

Similarly when canary varibale is enabled,we get different values **x=40 and y=7**.
In this case values are different because compiler does some optimization on allocation of local variables in the stack and compiler allocate memory in different manner and also canary variable is also in the stack.

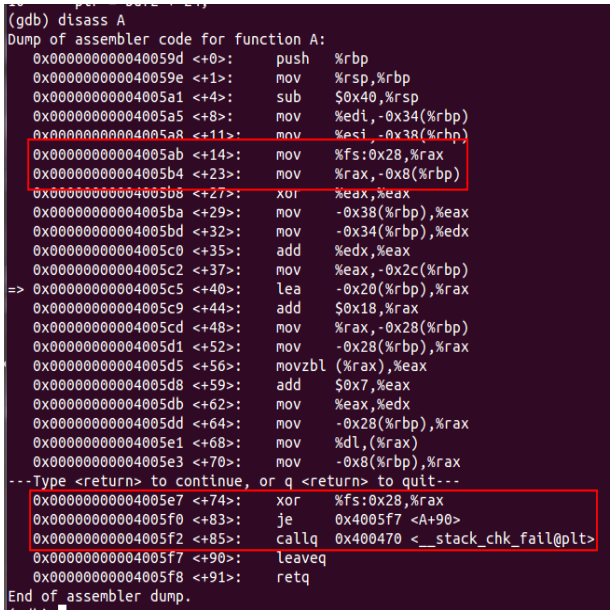

Figure 6: Canary varibale intialization and checking for buffer-overflow attack

3.2 On Windows(x86_64)

3.2.1 Canary variable is enabled

1. On windows machine stack word size is 4 bytes.

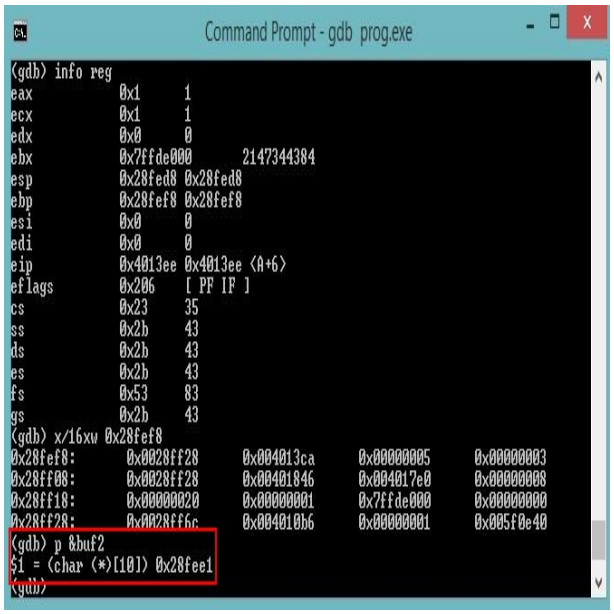

Figure 7: Address of buf2

2. We now check for return address of function A.

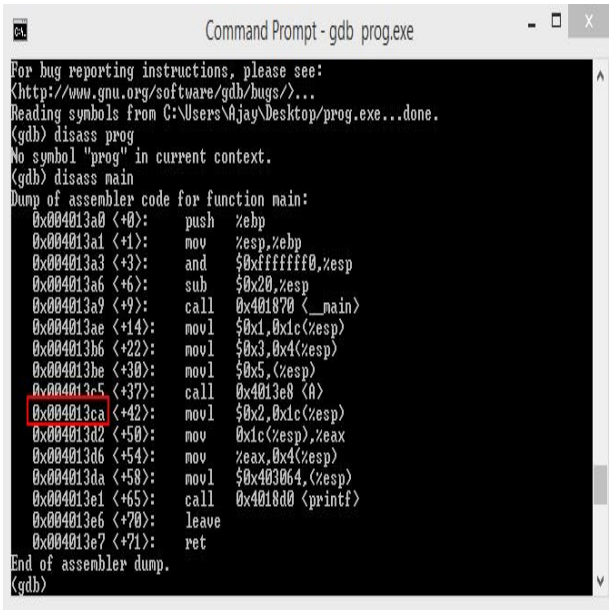

Figure 8: Return address of function A

3. Checking contents of register to get address of base-pointer of stack-frame of function A.

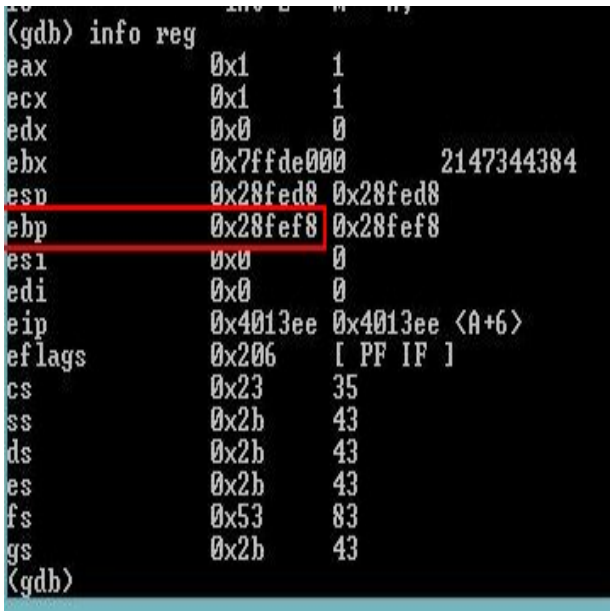

Figure 9: Address of base-pointer

(base-pointer +4) will give the location of return Address stored in stack frame.
So,**ox28fefc** is the location where return address is stored.
Therefore value of x=27(differnce between location of return address and address of buf2)
And value of y=8.

3.2.2 Canary variable is disabled

To disable the canary in windows machine we compile our program with command as :
`gcc -g -fstack-protector -o prog prog.c`

Similarly with stack-protection on we get **x=26 and y=8.**
This is because memory for local variables are allocated differently in presence of canary variable.

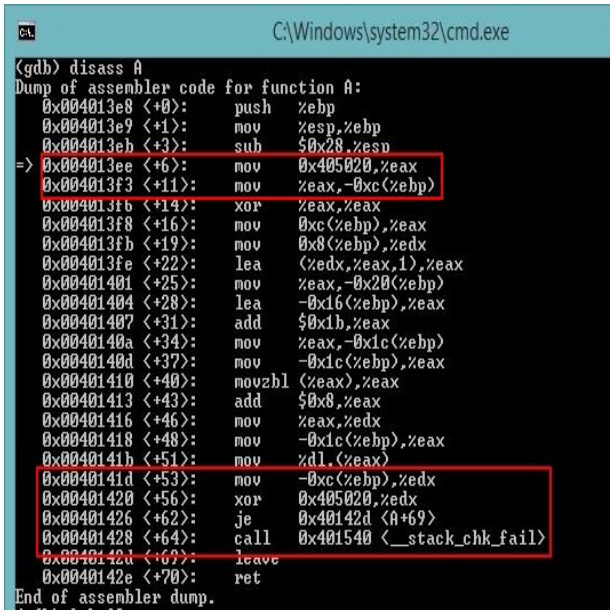

Figure 10: Canary varibale intialization and checking for buffer-overflow attack

Table 1: Values of x and y

linux(x_86_64) canary disabled	x=56	y=7
linux(x_86_64) canary enabled	x=40	y=7
windows(x_86_64) canary disabled	x=27	y=8
windows(x_86_64) canary enabled	x=26	y=8