

# CS 742 Foundations of Network Security and Cryptography

## Assignment 3

Ajay Kedare  
153059007

Astha Jada  
153050027

Neha Garg  
153050039

Sankalp Rangare  
153050087

- 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.  $A \rightarrow B : M, A, B, \{N_a, M, A, B\}K_{as}$
2.  $B \rightarrow KDC : M, A, B, \{N_a, M, A, B\}K_{as}, \{N_b, M, A, B\}K_{bs}$
3.  $KDC \rightarrow B : \{M, N_a, K_{a,b}\}K_{as}, \{M, N_b, K_{a,b}\}K_{bs}$
4.  $B \rightarrow A : \{N_a, K_{a,b}\}K_{as}, \{N_b\}K_{ab}$
5.  $A \rightarrow B : \{N_b - 1\}K_{ab}$

Here, M is session identifier.

$N_a$  and  $N_b$  are the nonces.

$A \rightarrow B : M, A, B, \{N_a, M, A, B\}K_{as}$

A sends encrypted data to KDC which includes nonce, session identifier and identities of A & B. A also sends this in plaintext to B.

$B \rightarrow KDC : M, A, B, \{N_a, M, A, B\}K_{as}, \{N_b, M, A, B\}K_{bs}$

B now communicates with KDC and forwards A's encrypted data along with encrypted data from B which includes B's nonce, session identifier and identities of A & B.

$KDC \rightarrow B : \{M, N_a, K_{a,b}\}K_{as}, \{M, N_b, K_{a,b}\}K_{bs}$

KDC, replies back with session key  $K_{ab}$  for both A & B which is encrypted with shared keys of A & B with KDC.

$B \rightarrow A : \{N_a, K_{a,b}\}K_{as}, \{N_b\}K_{ab}$

B sends the received ticket to A which has session key and also nonces encrypted with session key.

$A \rightarrow B : \{N_b\}K_{ab}$

A decrypts the session key from ticket and verifies the nonces by using session key.

This protocol provides key confirmation and entity authentication. A is sending nonce to B which is encrypted for KDC, and when A receives back the session key, it verifies the nonce with the nonce earlier sent to KDC. Hence any replay of message will be detected with the freshness of nonce.

Using timestamp we reduced the number of messages to 4 as below:

1.  $A \rightarrow B : A, N_a$
2.  $B \rightarrow KDC : B, N_b, \{A, N_a, T_b\}K_{bs}$
3.  $KDC \rightarrow A : \{B, N_a, K_{ab}, T_b\}K_{as}, \{A, K_{ab}, T_b\}K_{bs}, N_b$
4.  $A \rightarrow B : \{A, K_{ab}, T_b\}K_{bs}, \{N_b\}K_{ab}$

Here,  $T_b$  is timestamp which specifies the duration for which session is valid.

$N_a$  and  $N_b$  are the nonces.

$A \rightarrow B : A, N_a$

First, A sends his identification and nonce to B.

$B \rightarrow KDC : B, N_b, \{A, N_a, T_b\}K_{bs}$

B forwards this nonce and A's identification to KDC requesting for the ticket. which is encrypted with  $K_{bs}$ . B also sends his nonce so that he can verify when he receives the actual session key. A timestamp is added that decides the duration of session.

$KDC \rightarrow A : \{B, N_a, K_{ab}, T_b\}K_{as}, \{A, K_{ab}, T_b\}K_{bs}, N_b$

KDC, now generates the session key  $K_{ab}$  and forward it to A along with A's nonce, timestamp  $T_b$  encrypted with  $K_{as}$ . It also includes the ticket which has the session key, timestamp and will be forwarded to B.

$A \rightarrow B : \{A, K_{ab}, T_b\}K_{bs}, \{N_b\}K_{ab}$

A forwards the ticket to B which has session key, timestamp and encrypts the B's nonce with session key  $K_{ab}$

Here B receives the ticket from A and will verify the timestamp whether the ticket is valid for that duration or not hence any replay attack is not possible which is outside the valid duration.

## 2 Study the following attacks on SSL or on OpenSSL – BEAST, Heartbleed, Fluke. For each attack answer the following: What does the attack accomplish? Describe 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 mode 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 encryption. 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 Describe a “typical” attack scenario.

Considering that attacker has identified a block of ciphertext  $C_i$  that he/she wants to decrypt, because it contains the session cookie. Now as he already sniffed the network so he already has  $C_{i-1}$  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 until 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 mode 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 encrypted. 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 send 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 Describe 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 Describe 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 variables 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 variables.

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:
0x00000000400562 <+0>:    push    %rbp
0x00000000400563 <+1>:    mov     %rsp,%rbp
0x00000000400566 <+4>:    sub     $0x10,%rsp
0x0000000040056a <+8>:    movl    $0x1,-0x4(%rbp)
0x00000000400571 <+15>:   mov     $0x3,%esi
0x00000000400576 <+20>:   mov     $0x5,%edi
0x0000000040057b <+25>:   callq   0x40052d <A>
0x00000000400580 <+30>:   movl    $0x2,-0x4(%rbp)
0x00000000400587 <+37>:   mov     -0x4(%rbp),%eax
0x0000000040058a <+40>:   mov     %eax,%esi
0x0000000040058c <+42>:   mov     $0x400624,%edi
0x00000000400591 <+47>:   mov     $0x0,%eax
0x00000000400596 <+52>:   callq   0x400410 <printf@plt>
0x0000000040059b <+57>:   leaveq  %eax
0x0000000040059c <+58>:   retq
End of assembler dump.
(gdb)
```

Figure 2: Return address of function `A()`

### 3.Stack layout:

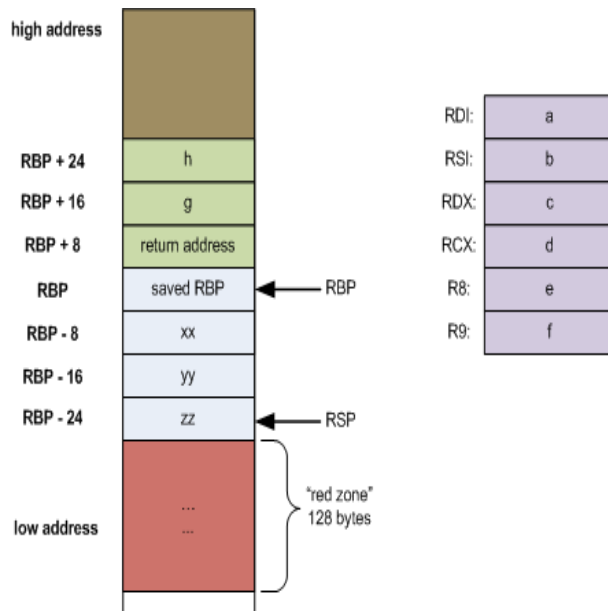


Figure 3: Stack frame layout

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

```
(gdb) info reg
rax      0x8      8
rbx      0x0      0
rcx      0x0      0
rdx      0x5      5
rsi      0x3      3
rdi      0x5      5
rbp      0x7fffffffdf00  0x7fffffffdf00
rsp      0x7fffffffdf00  0x7fffffffdf00
r8       0x7ffff7dd4e80  140737351863936
r9       0x7ffff7dea560  140737351951712
r10      0x7fffffffddb0  140737488346544
r11      0x7ffff7a36dd0  140737348070864
r12      0x400440  4195392
r13      0x7fffffffef00  140737488347136
r14      0x0      0
```

Figure 4: Base-pointer address

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

```
(gdb) x/1xw 0x7fffffffdf08
0x7fffffffdf08: 0x00400580
(gdb)
```

Figure 5: Location of return address in stack-frame

Now, we have the location of **return address** and address of `buf2`. We calculated the difference 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 variable 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 allocates memory in different manner and also canary variable is also in the stack.

```
(gdb) disass A
Dump of assembler code for function A:
0x0000000040059d <+0>:  push    %rbp
0x0000000040059e <+1>:  mov     %rsp,%rbp
0x000000004005a1 <+4>:  sub     $0x40,%rsp
0x000000004005a5 <+8>:  mov     %edi,-0x34(%rbp)
0x000000004005a8 <+11>: mov     %esi,-0x38(%rbp)
0x000000004005ab <+14>: mov     %fs:0x28,%rax
0x000000004005b4 <+23>: mov     %rax,-0x8(%rbp)
0x000000004005b6 <+27>: xor     %eax,%eax
0x000000004005ba <+29>: mov     -0x38(%rbp),%eax
0x000000004005bd <+32>: mov     -0x34(%rbp),%edx
0x000000004005c0 <+35>: add     %edx,%eax
0x000000004005c2 <+37>: mov     %eax,-0x2c(%rbp)
=> 0x000000004005c5 <+40>: lea     -0x20(%rbp),%rax
0x000000004005c9 <+44>: add     $0x18,%rax
0x000000004005cd <+48>: mov     %rax,-0x28(%rbp)
0x000000004005d1 <+52>: mov     -0x28(%rbp),%rax
0x000000004005d5 <+56>: movzbl (%rax),%eax
0x000000004005d8 <+59>: add     $0x7,%eax
0x000000004005db <+62>: mov     %eax,%edx
0x000000004005dd <+64>: mov     -0x28(%rbp),%rax
0x000000004005e1 <+68>: mov     %dl,(%rax)
0x000000004005e3 <+70>: mov     -0x8(%rbp),%rax
---Type <return> to continue, or q <return> to quit---
0x000000004005e7 <+74>: xor     %fs:0x28,%rax
0x000000004005f0 <+83>: je      0x4005f7 <A+90>
0x000000004005f2 <+85>: callq   0x400470 <__stack_chk_fail@plt>
0x000000004005f7 <+90>: leaveq
0x000000004005f8 <+91>: retq
End of assembler dump.
```

Figure 6: Canary variable initialization 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.



```

Command Prompt - gdb prog.exe

(gdb) info reg
eax      0x1      1
ecx      0x1      1
edx      0x0      0
ebx      0x7ffde000      2147344304
esp      0x28fef8      0x28fef8
ebp      0x28fef8      0x28fef8
esi      0x0      0
edi      0x0      0
eip      0x4013ee      0x4013ee <A+6>
eflags   0x206      [ PF IF ]
cs       0x23      35
ss       0x2b      43
ds       0x2b      43
es       0x2b      43
fs       0x53      83
gs       0x2b      43

(gdb) x/16xu 0x28fef8
0x28fef8:  0x0028ff28      0x004013ca      0x00000005      0x00000003
0x28ff08:  0x0028ff28      0x00401846      0x004017e0      0x00000008
0x28ff18:  0x00000020      0x00000001      0x7ffde000      0x00000000
0x28ff28:  0x0028ffb6      0x004010b6      0x00000001      0x005f0e40

(gdb) p buf2
$1 = (char (*)[10]) 0x28fee1
(gdb)

```

Figure 7: Address of buf2

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

```

Command Prompt - gdb prog.exe

For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from C:\Users\Ajay\Desktop\prog.exe...done.
(gdb) disass prog
No symbol "prog" in current context.
(gdb) disass main
Dump of assembler code for function main:
0x004013a0 <+0>:  push    %ebp
0x004013a1 <+1>:  mov     %esp,%ebp
0x004013a3 <+3>:  and     $0xffffffff,%esp
0x004013a6 <+6>:  sub     $0x20,%esp
0x004013a9 <+9>:  call    0x401870 <_main>
0x004013ae <+14>:  movl    $0x1,0x1c(%esp)
0x004013b6 <+22>:  movl    $0x3,0x4(%esp)
0x004013be <+30>:  movl    $0x5,%esp
0x004013c5 <+37>:  call    0x4013e8 <A>
0x004013ca <+42>:  movl    $0x2,0x1c(%esp)
0x004013d2 <+50>:  mov     0x1c(%esp),%eax
0x004013d6 <+54>:  mov     %eax,0x4(%esp)
0x004013da <+58>:  movl    $0x403064,%esp
0x004013e1 <+65>:  call    0x4018d0 <printf>
0x004013e6 <+70>:  leave   0
0x004013e7 <+71>:  ret

End of assembler dump.
(gdb)

```

Figure 8: Return address of function A

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

```

(gdb) info reg
eax          0x1      1
ecx          0x1      1
edx          0x0      0
ebx          0x7ffde000 2147344384
esp          0x28fed8 0x28fed8
ebp          0x28fef8 0x28fef8
esi          0x0      0
edi          0x0      0
eip          0x4013ee 0x4013ee <A+6>
eflags      0x206    [ PF IF ]
cs           0x23     35
ss           0x2b     43
ds           0x2b     43
es           0x2b     43
fs           0x53     83
gs           0x2b     43
(gdb)

```

Figure 9: Address of base-pointer

(base-pointer +4) will give the location of return Address stored in stack frame.  
 So, **0x28fefe** is the location where return address is stored.  
 Therefore value of x=27(difference 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.

```

C:\Windows\system32\cmd.exe

(gdb) disass A
Dump of assembler code for function A:
0x004013e8 <+0>:      push    %ebp
0x004013e9 <+1>:      mov     %esp,%ebp
0x004013eb <+3>:      sub     $0x28,%esp
=> 0x004013ee <+6>:      mov     0x405020,%eax
0x004013f3 <+11>:     mov     %eax,-0xc(%ebp)
0x004013f6 <+14>:     xor     %eax,%eax
0x004013f8 <+16>:     mov     0xc(%ebp),%eax
0x004013fb <+19>:     mov     0x8(%ebp),%edx
0x004013fe <+22>:     lea     (%edx,%eax,1),%eax
0x00401401 <+25>:     mov     %eax,-0x20(%ebp)
0x00401404 <+28>:     lea     -0x16(%ebp),%eax
0x00401407 <+31>:     add     $0x1b,%eax
0x0040140a <+34>:     mov     %eax,-0x1c(%ebp)
0x0040140d <+37>:     mov     -0x1c(%ebp),%eax
0x00401410 <+40>:     movzbl (%eax),%eax
0x00401413 <+43>:     add     $0x8,%eax
0x00401416 <+46>:     mov     %eax,%edx
0x00401418 <+48>:     mov     -0x1c(%ebp),%eax
0x0040141b <+51>:     mov     %dl,(%eax)
0x0040141d <+53>:     mov     -0xc(%ebp),%edx
0x00401420 <+56>:     xor     0x405020,%edx
0x00401426 <+62>:     je      0x40142d <A+69>
0x00401428 <+64>:     call    0x401540 <__stack_chk_fail>
0x0040142d <+69>:     leave
0x0040142e <+70>:     ret
End of assembler dump.

```

Figure 10: Canary variable initialization 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