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

Security definitions

1.1 Exercise 1

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 2

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 3

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 4

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 5

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 crowd sourced 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 crowd sourced 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 crowd sourced 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.
   - Multi-factor authentication: implementing multi-factor authentication, such as using passwords and biometric factors, can ensure only authorized users access personal information and location data.

CHAPTER 2

Cryptography

2.1 Exercise 1

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.

2.2 Exercise 2

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 $2^{84-(2^{14}+20\cdot 2^{28}+30\cdot 2^{42})} < 2^{84-(32\cdot 2^{42})} = 2^{84-(2^{47})}$, which is still way larger than 2^{83}, thus definitely large enough.

CHAPTER 3

Binary vulnerabilities

3.1 Exercise 1

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:

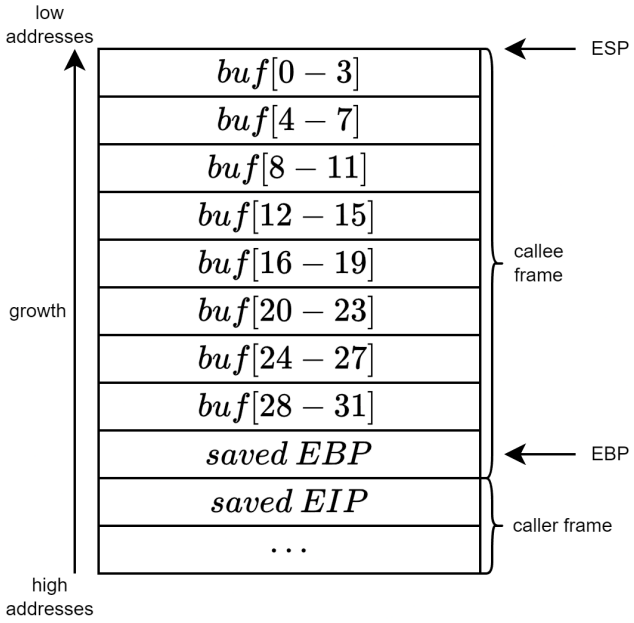
   - Direction of growth and high-low addresses.
   - The name of each allocated variable.
   - The boundaries of frame of the function frames (`main` and `vuln`).

   Show also the content of the caller frame.

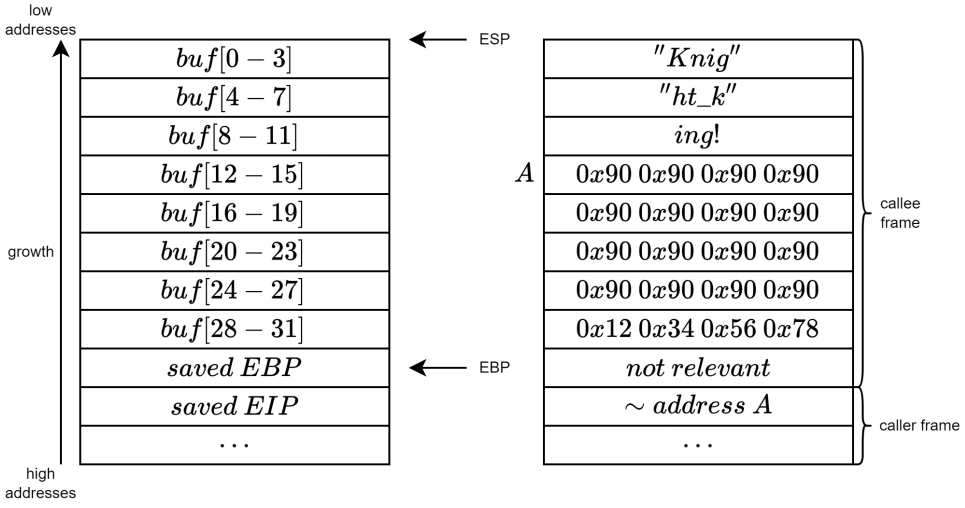
2. Write an exploit for the buffer overflow vulnerability in the above program. Your exploit should execute the following simple shell code, composed only by four instructions: `0x12 0x34 0x56 0x78`. Write clearly all the steps and assumptions you need for the exploitation, and show the stack layout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

3.2 Exercise 2

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.

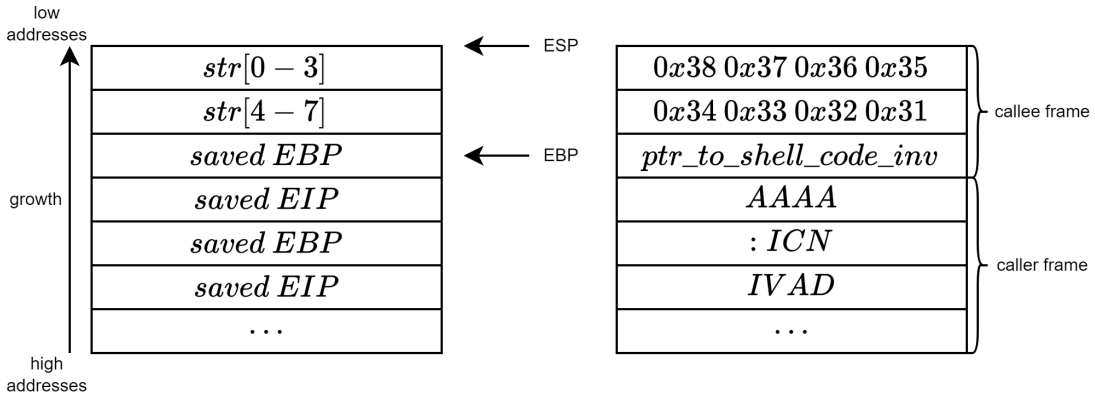
2. The input required to exploit the vulnerability is as follows:

   ```
   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:



3.3 Exercise 3

Consider the C program below:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  void vuln(int n) {
6          struct {
7          char buf[16];
8          char tmp[16];
9          int sparrow = 0xBAAAAAAD;
10     } s;
11
12     if (n > 0 && n < 16) {
13         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.

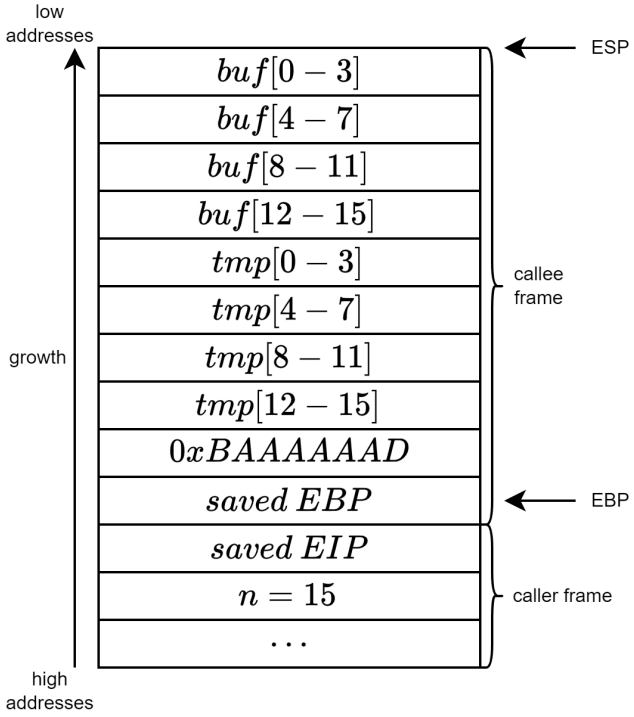
3. Write an exploit for the buffer overflow vulnerability in the above program to execute the following simple shell code, composed only by four instructions: `0x51 0x52 0x5a 0x5b`. Make sure that you show how the exploit will appear in the process memory with respect to the stack layout right before 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.

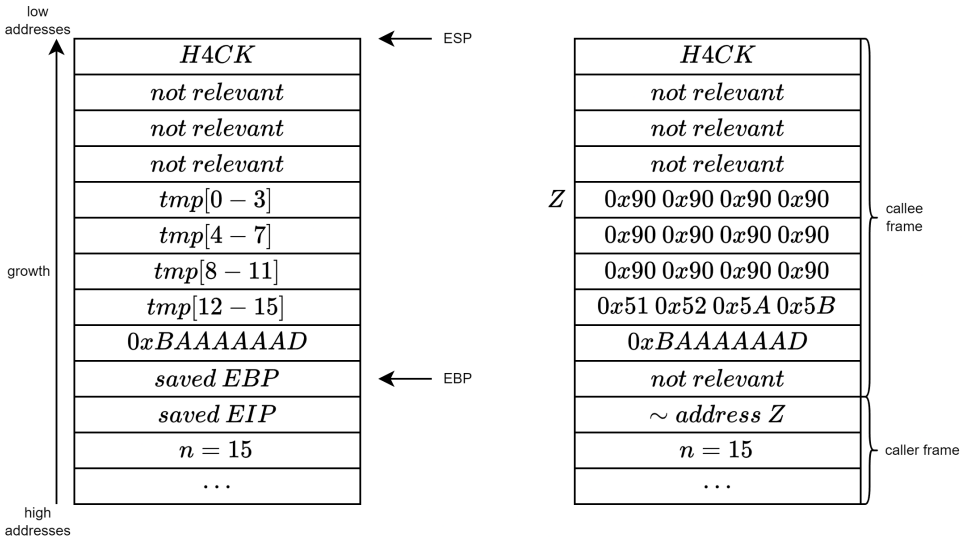
Solution

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:



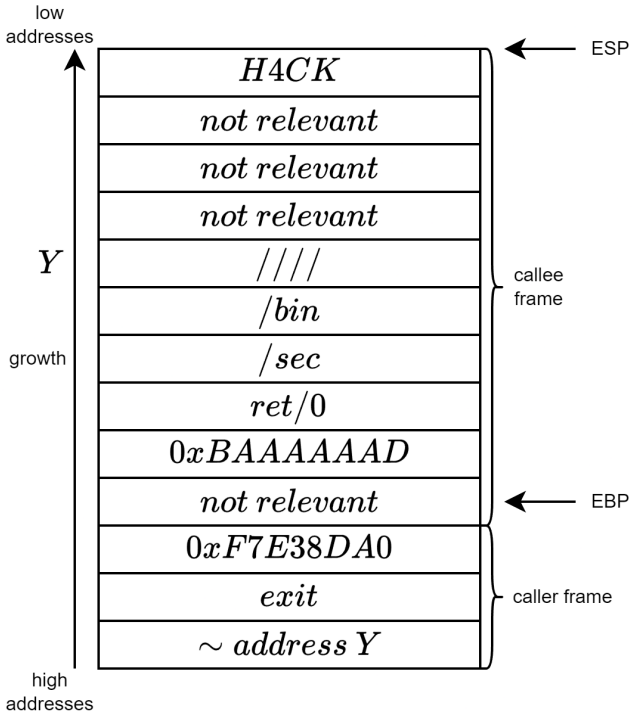
2. The buffer overflow occurs at line seventeen because `scanf` reads an entire string without considering the size of the `buf` buffer. Conversely, at line thirteen, there's no buffer overflow since the value of n is guaranteed to be smaller than the buffer's size.

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);
```

3.4 Exercise 4

Consider the C program below.

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int guess(char *user) {
6      struct {
7          int n;
8          char usr[16];
9          char buf[16];
10     } s;
11
12     snprintf(s.usr, 16, "%s", user);
13
14     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`)

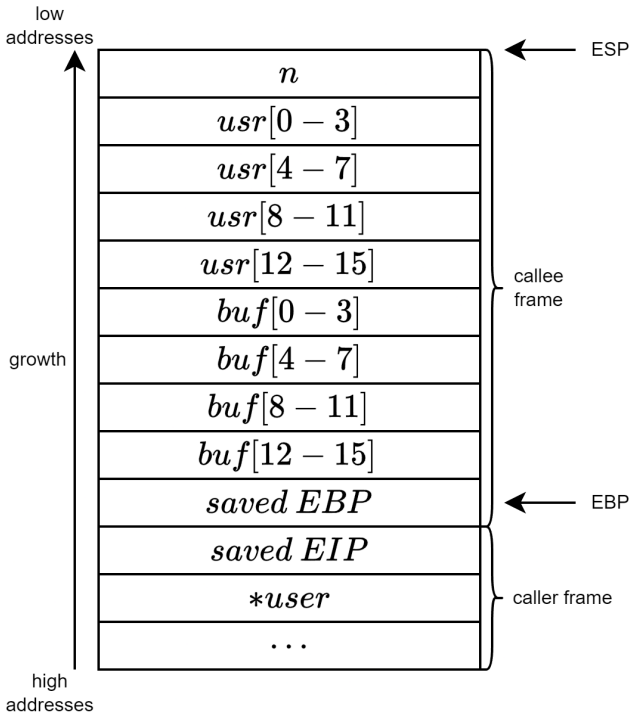
   Show also the content of the caller frame (you can ignore the environment variables: just focus on what matters for the exploitation of typical memory corruption vulnerabilities). Assume that the program has been properly invoked with a single command line argument.

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).

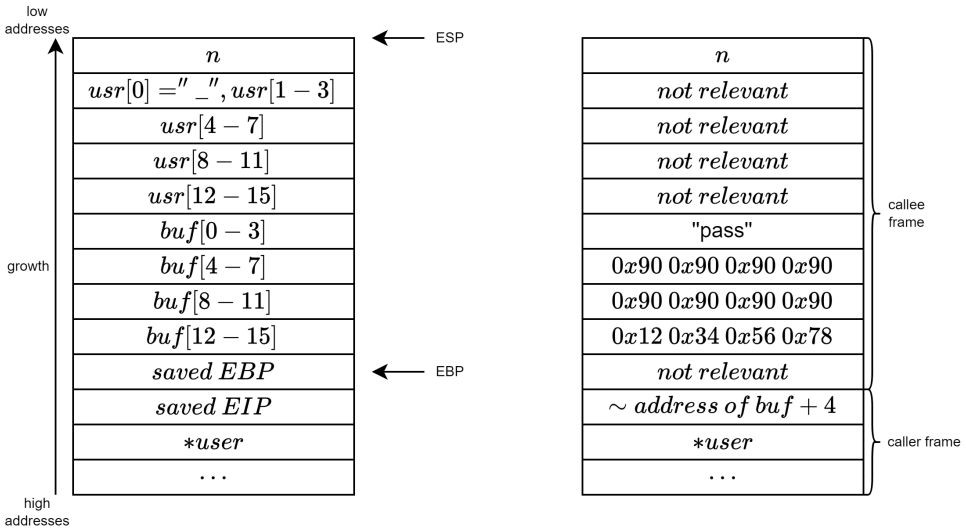
Solution

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:



3.5 Exercise 5

Consider the C program below:

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

```
2  #include <stdlib.h>
3  #include <string.h>
4  #include <unistd.h>
5  #include <fcntl.h>
6
7  typedef struct {
8      char username[40];
9      char password[20];
10 } user_t;
11
12 void welcome_user(user_t * user){
13     char message[52];
14     strcpy(message, "Ciao to "); // len 8
15     strcat(message, user->username); // len 8+40
16     strcat(message, "\n"); // len 8+40+1 == 49 < 52, ok
17     printf(message);
18     return;
19 }
20
21 int main() {
22     user_t user;
23
24     printf("Insert your name: ");
25     read(0, user.username, 40); // 0 == stdin
26     printf("Insert your password: ");
27     read(0, user.password, 20); // 0 == stdin
28
29     welcome_user(&user);
30
31 return 0;
32 }
```

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.

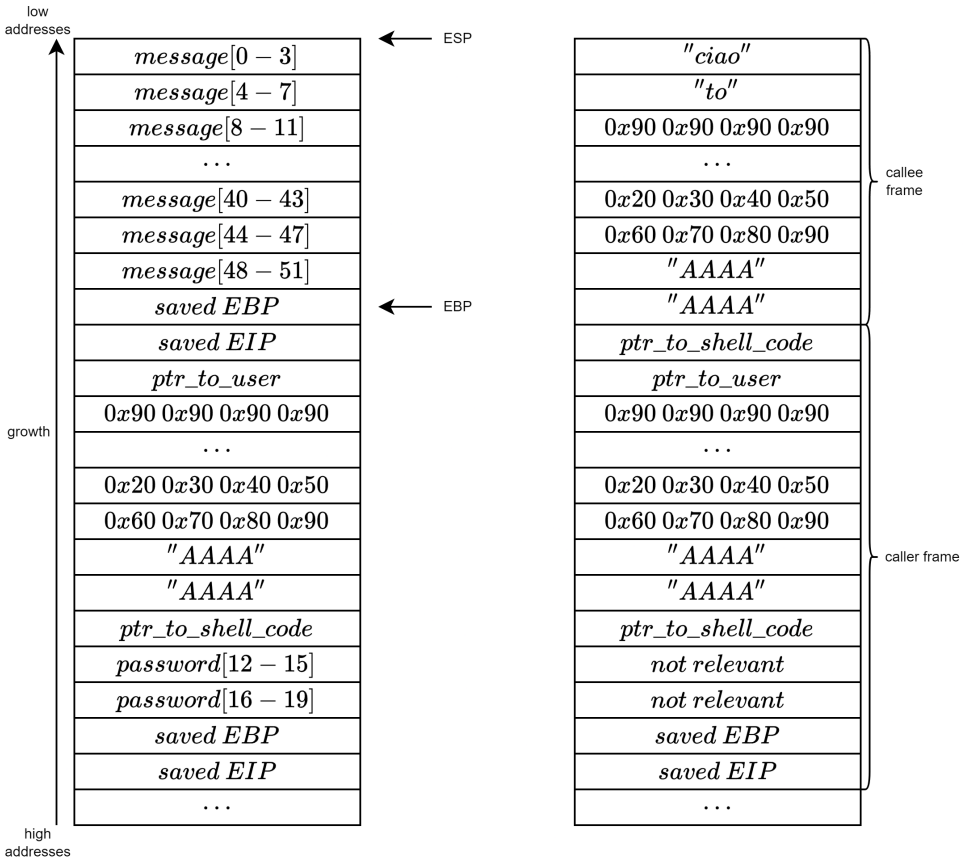
Solution

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:



3.6 Exercise 6

Consider the following code:

```
1  #include <stdio.h>
2
3  typedef struct
4  {
5      char buf2[15];
6      char buf[64];
7  } data_t;
8  // This function writes 5 times 15 char in data.buf
9  int scramble(int *sequence)
10 {
11     data_t data;
12
13     for (int i = 0; i < 5; i++)
14     {
15         scanf("\%15s", data.buf2);
16         strncpy(&data.buf[sequence[i]*15], data.buf2, 15);
17         printf(data.buf);
18     }
19 }
20
21 void main()
22 {
23     int sequence[5] = {3,1,4,0,2};
24
25     scramble(sequence);
26 }
```

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:

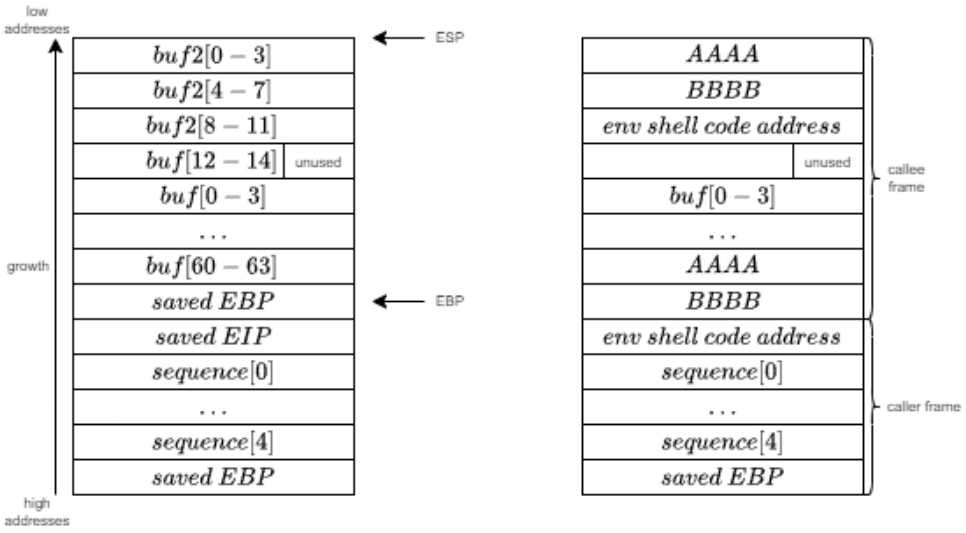
```
void main()
{
    int sequence[5];
    srand(0); // seed the random number generator with seed=0

    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.

3.7 Exercise 7

Consider the following code:

```
#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;

    s.p = s.sec1;
    s.backdoor = 0;

    printf("Please insert the encryption secret\n");
    scanf ("\%s", s.sec2);
    scanf ("\%7s", s.p);
    if (strncmp(s.sec2, "FL4G", 4) == 0) {
        encrypt(s.backdoor);
        return 1;
    }
    printf("To be, or not to be, that is the question...\n");
    abort ();
}

void main (int argc, char **argv){
    verify();
}
```

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

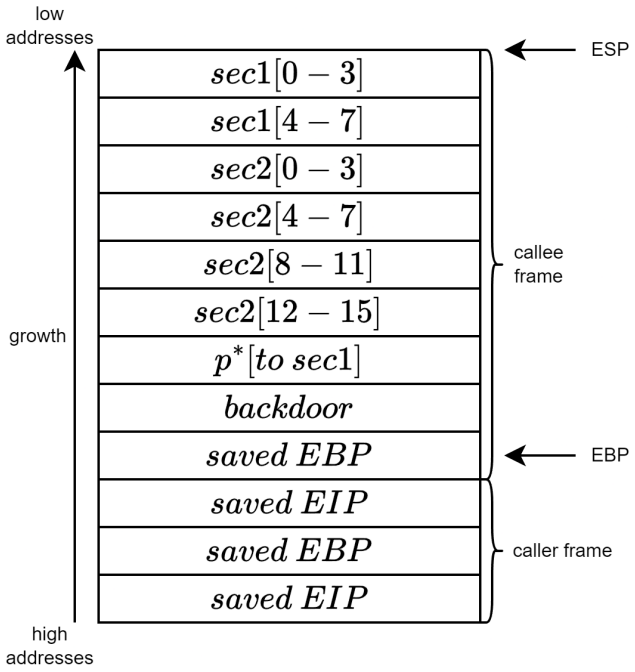
   Knowing that dec(0xf4d0) = 62672 and dec(0xe349) = 58185, write the exploit clearly, detailing all the components of the format string and all the steps that lead to a successful exploitation.

Solution

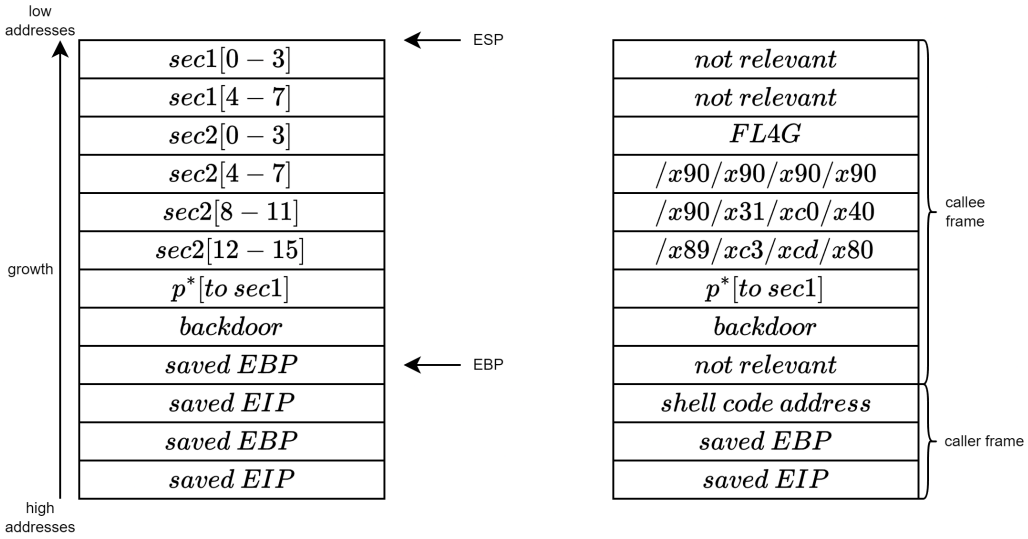
1. There is a buffer overflow ar line 37 because it reads an entire string without considering the length.

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 4

Web vulnerabilities

4.1 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:

```
1  def missing_file(cookie, reqpath):
2  print "HTTP/1.0 200 OK"
3      print "Content-Type: text/html"
4      print ""
5      user = check_cookie(cookie)
6      if user is None:
7          print "Please log in first"
8          return
9
10     print "<p>We are sorry, but the server could not locate file" + reqpath
11     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:

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

```
10      # 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.

4.2 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:

```
1  def ship_stuff(request):
2      # code to send HTTP header (not relevant)
3      user = check_cookie(request.cookie)
4      if user is None:
5          print "Please log in first"
6          return
7
8      product_id = request.params['product_id'] # GET parameter
9      receiver_id = request.params['receiver_id'] # GET parameter
10
11     query1 = 'SELECT p_id, product_name FROM warehouse, ownership WHERE p_id = ' + pr
12     db.execute(query)
13     row = db.fetchone()
14     if row is None:
15         print "Product", product_id, "is not existent"
16         return
17
18     query2 = 'SELECT u_id, username FROM accounts WHERE u_id = ' + receiver_id + ';'
19     db.execute(query)
20     row = db.fetchone()
21     if row is None:
22         print "User", receiver_id, "is not existent"
23         return
24     # 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';--
```

4.3 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

```
1  var id = request.get['id'];
2  var prep_query = prepared_statement("SELECT username FROM users WHERE id=? LIMIT 1");
3  var username = query(prep_query, id);
4  var prep_query = prepared_statement("SELECT * FROM comments WHERE username=?");
5  var comments = query(prep_query, username);
6  for comment in comments {
7      echo htmlentities(comment);
8  }
```

Login

```
1  var password = md5(request.post['password']);
2  var username = request.post['username'];
3  var prep_query = prepared_statement("SELECT username FROM users WHERE username=? AND
4  var result = query(prep_query, username, password);
5  if (result) {
6      session.set('username', username);
7      echo "Logged in.";
8  } else {
9      echo "User" + username + "does not exists!";
10 }
```

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., <, >, " and ' 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).

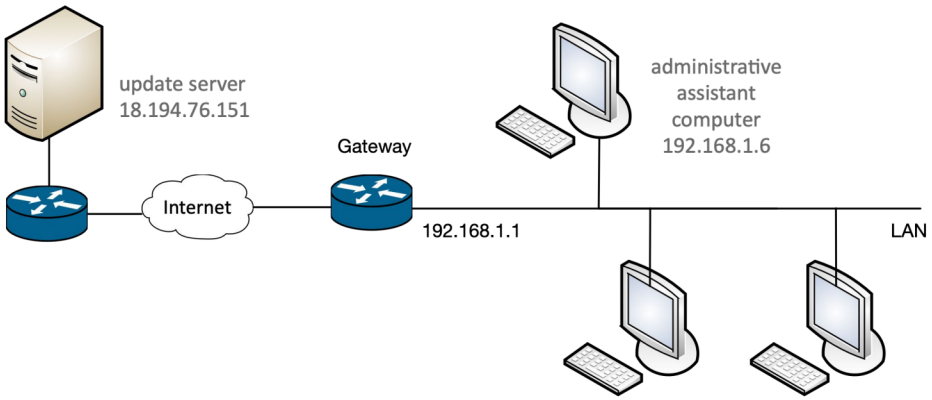
CHAPTER 5

Network security

5.1 Exercise 1

As part of your job as a security analyst, one of your clients discovers that their network is compromised. In particular, from an early analysis, they have ground to suspect that the start of the compromise was a network attack against the computer of the administrative assistant.

Consider the following (simplified) schema of the company network.



Your client managed to capture the network traffic on the administrative assistant's computer (IP address 192.168.1.6 and MAC address dc:a9:04:7a:ce:29) when the attack was taking place. During the traffic capture, the computer was automatically updating a well-known accounting software from the software vendor's web server (IP address 18.194.76.151 and MAC address dc:a6:03:01:02:fe). You also know that the IP address of the LAN interface of the company's network gateway is 192.168.1.1, and its MAC address is b6:28:97:ca:b7:48.

- dc:a9:04:7a:ce:29 → ff:ff:ff:ff:ff:ff ARP Who has 192.168.1.1? Tell 192.168.1.6
- 38:60:77:b9:79:98 → dc:a9:04:7a:ce:29 ARP 192.168.1.1 is at 38:60:77:b9:79:98
- b6:28:97:ca:b7:48 → dc:a9:04:7a:ce:29 ARP 192.168.1.1 is at b6:28:97:ca:b7:48
- 192.168.1.6 (dc:a9:04:7a:ce:29) → 18.194.76.151 (38:60:77:b9:79:98) TCP SYN
- 18.194.76.151 (38:60:77:b9:79:98) → 192.168.1.6 (dc:a9:04:7a:ce:29) TCP SYN, ACK

- 38:60:77:b9:79:98 → dc:a9:04:7a:ce:29 ARP 192.168.1.1 is at 38:60:77:b9:79:98
- 192.168.1.6 (dc:a9:04:7a:ce:29) → 18.194.76.151 (38:60:77:b9:79:98) TCP ACK
- 38:60:77:b9:79:98 → dc:a9:04:7a:ce:29 ARP 192.168.1.1 is at 38:60:77:b9:79:98
- 192.168.1.6 (dc:a9:04:7a:ce:29) → 18.194.76.151 (38:60:77:b9:79:98) TCP HTTP GET /downloads/software-update.exe
- 18.194.76.151 (38:60:77:b9:79:98) → 192.168.1.6 (dc:a9:04:7a:ce:29) TCP HTTP 200 OK ...

1. Describe the attack going on in the network, specifying the name and providing a short explanation of how the attack works in general.
2. What is the goal of the attack, in this specific case? Motivate your answer.
3. Can you tell the IP address of the attacker? And the MAC address?
4. Given only the above packet capture, can you tell whether the attacker is located (i.e., on the LAN, on the same network of the web server, or on an arbitrary Internet-connected network)? Why?

Solution

1. The attack is an ARP spoofing. It is a type of attack in which a malicious actor sends falsified ARP (Address Resolution Protocol) messages over a LAN.
2. Sniffing or manipulation of the traffic to or from the compromised machine. We can rule out DOS as the traffic passes (there are responses from the server). Likely, given the scenario (malware infection), the attack is targeted at tampering with the data in transit rather than (or in addition to) sniffing.
3. We cannot tell the real IP of the attacker, but the MAC is 38:60:77:b9:79:98 (that could be also spoofed).
4. The attacker is located on the same network of the target machine, i.e., on the LAN.

5.2 Exercise 2

Suppose that you are the network security administrator of the network 131.168.0.0/24, with gateway 131.168.0.1 and DNS servers 131.168.0.100 and 131.168.0.101. While examining the network activity, you notice a DHCP offer packet coming from IP 131.168.0.10 with gateway set to 131.168.0.5. Answer the following questions and provide a reason. Answers with no reason will not give any point.

1. What kind of attack do you suspect, and how does it work?
2. Why such an attack works?
3. Can you tell the IP address of the host where those packets come from?
4. Can you tell the IP address or network address of the victim?

Solution

1. DHCP poisoning. Someone is trying to trick a client connected to 131.168.0.0/24 into believing that 131.168.0.5 is the gateway, by sending a crafted DHCP offer before, which comes before the real offer sent out by the real DHCP server.

2. Because the DHCP protocol does not support authentication, so the client must blindly believe any DHCP offer that it sees, and because an arbitrary client can race (and win) against the real DHCP server

3. Not really. 131.168.0.10 is the sender of the DHCP offer, but the address may be spoofed. We could look at the MAC address of the sender, but it could be spoofed as well.

4. From the above information there is little evidence to say that, although the potential victims are those that will receive and accept the spoofed DHCP offers. So, likely, 131.168.0.0/24

5.3 Exercise 3

A network analyst is analyzing some traffic captured from a network belonging to Politecnico di Milano. The network is: 131.175.0.0/16. In particular, we suspect that a database server, whose IP address is 131.175.14.12, is victim of an attack. Indeed, observing the network traffic, we notice the following pattern:

- 131.175.14.12 → 131.175.255.255 [ICMP] Echo (ping) request (broadcast)
- 131.175.0.2 → 131.175.14.12 [ICMP] Echo (ping) reply
- 7a:ce:29 → ff:ff:ff (broadcast) [ARP] Who has 131.175.14.12? Tell 131.175.0.2
- 4b:74:28 → 7a:ce:29 [ARP] 131.175.14.12 is at 4b:74:28
- 131.175.0.3 → 131.175.14.12 [ICMP] Echo (ping) reply
- 131.175.0.4 → 131.175.14.12 [ICMP] Echo (ping) reply ⋮
- 131.175.255.251 → 131.175.14.12 [ICMP] Echo (ping) reply
- 131.175.255.252 → 131.175.14.12 [ICMP] Echo (ping) reply
- 131.175.255.253 → 131.175.14.12 [ICMP] Echo (ping) reply

1. Describe what attack you think is going on, and what is the feature (or lack thereof) of the involved protocol(s) that enable this attack.

2. Describe what you think is the concrete goal(s) of the attack in this scenario.

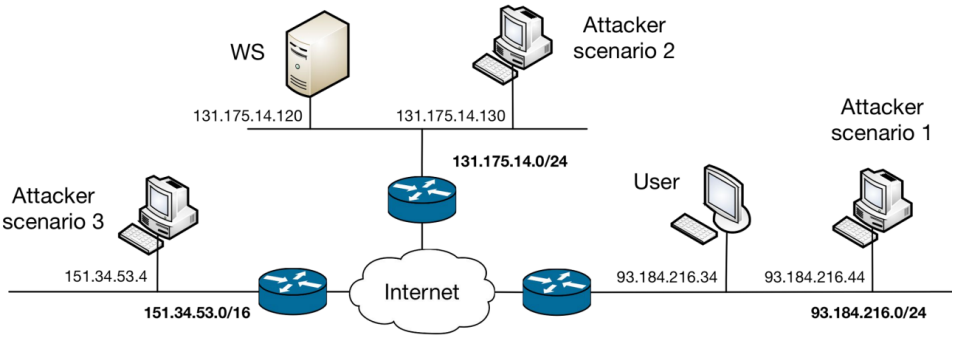
3. Can you tell the IP and MAC address of the attacker? Why?

Solution

1. PING Smurf.
2. The goal is to saturate the resources of the victim using other machines on the network as an amplification mean. This results in a denial of service.
3. No, the IP address is spoofed for sure, and the MAC address can be spoofed as well.

5.4 Exercise 4

Consider the following network diagram:



A user, with IP address 93.184.216.34, is attempting to download a software executable from a web server in the Politecnico di Milano network with IP address 131.175.14.120, over the HTTP protocol (no HTTPS, no signatures, nothing). Assume that the user's browser already cached the IP address of downloads.polimi.it (i.e., it does not perform any DNS request), and that there is no firewall involved. An attacker, who knows that the user is about to download this software, wants to target our user by carrying out an attack to replace the downloaded software with a piece of malware. For each of the following attack scenarios, state whether the attacker is able to fulfill his goals. If you deem it possible, describe an attack that allows to do so: state the name of the class of attacks, and describe all the steps needed to make it work in this specific scenarios. If multiple classes of attacks are possible, focus on the simplest one that gets the job done. If no attack is possible, please explain why.

1. Attacker: 93.184.216.44; same network and broadcast domain of the user.
2. Attacker: 131.175.14.130; same network of web server, but different than user.
3. Attacker: 151.34.53.4; attacker, user and web server on three different networks.

For the next questions consider ONLY scenario 3. Assume that each involved computer and server implements a custom TCP/IP stack that, for performance reasons, sets the TCP initial sequence number in the SYN and SYN+ACK packets as the most significant bits of the current timestamp.

3. Describe the security issue with the proposed ISN implementation, and propose a way to solve the issue.
4. Describe how the attacker can perform the above attack, this time exploiting the security issues raised by the custom ISN implementation. Describe all the steps and assumptions that you need to perform this attack.

5. Assume you are the network security administrator of the network of the attacker (in the scenario 3 and assuming the custom TCP initial sequence number implementation), and that you control the border router between the network 151.34.53.0/24 and the rest of the Internet. Propose a way to prevent the attack. Can the administrator of the other two border routers in the diagram deploy the same mitigation, and obtain the same result? Why?

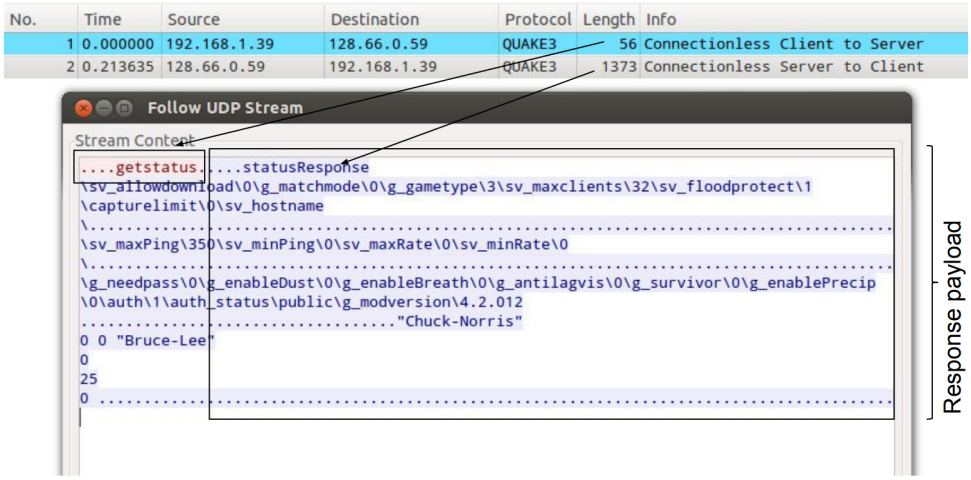
Solution

1. ARP spoofing.
2. ARP spoofing (but this time target the server).
3. No attack possible, because HTTP uses TCP and, if sequence numbers are correctly implemented, not possible to perform TCP hijacking. Also, DNS poisoning not possible as DNS response already cached.
4. Can predict ISN, so we can solve by using random ISN.
5. TCP hijacking. We can guess the ISN of the SYN packet sent by the victim (the user), we can spoof the web server IP and send a correct SYN+ACK to it and, if we can guess the content of the request (we do), subsequent packets. This way we can send a different payload. In parallel we can also use TCP hijacking to send a fake RST to the actual server spoofing the user's IP address. Problem and assumption: we need to know the ephemeral port used by the client to initiate the connection.
6. We can filter the packets with source IPs not belonging to our network. Other routers can't do this, they can only filter out packets coming from outside with spoofed IPs belonging to their network, but it's of little use in this scenario.

5.5 Exercise 5

Consider the Quake III Arena Network Protocol, a stateless client-server protocol used by the classic multi-player game. Quake III supports multiple name servers (indeed, anyone can run their server, expose it to the whole Internet, and even have it indexed by the "master server" for easy discovery). When a client connects to one of the many available servers, it needs to retrieve some information. To this purpose, the Quake III protocol implements the command gestatus, accessible without authentication. When the server receives this command (via an UDP packet, destination port 27960), it replies with various information, such as: the list of enabled options, the hostname, the number of connected clients, and the type of supported game.

The image below is an example of protocol exchange, as shown in Wireshark, a packet capture tool: a client (192.168.1.39) sends a getstatus message to a server (128.66.0.59); the server replies with a statusResponse message, containing the information about the server in its payload.



1. The part of the Quake III Arena protocol described in the previous page can be misused to ease a DoS attack against a victim. Please explain how an attack can misuse this protocol for this purpose, showing a concrete scenario where an attacker (who controls the server with IP address 93.184.216.32) aims to launch a DoS against the IP address 131.175.14.19. Make sure you mention in your answer the feature of this protocol that allows the misuse for DoS purpose.

We discovered in the wild a DoS attack that exploits this protocol. The network administrators of the company that was hit by this attack were able to capture the following headers of some suspect packets at their network's border firewall:

- IP 87.98.244.20 (src port 27960) ¿ 104.28.1.1 (dst port 12345) UDP, length 1373
- IP 87.98.244.20 (src port 27960) ¿ 104.28.1.1 (dst port 12345) UDP, length 1373
- IP 188.138.125.254 (src port 27960) ¿ 104.28.1.1 (dst port 32451) UDP, length 1400
- IP 188.138.125.254 (src port 27960) ¿ 104.28.1.1 (dst port 32451) UDP, length 1400
- IP 188.138.125.254 (src port 27960) ¿ 104.28.1.1 (dst port 32451) UDP, length 1400
- IP 188.138.125.254 (src port 27960) ¿ 104.28.1.1 (dst port 32451) UDP, length 1400
- IP 188.138.125.254 (src port 27960) ¿ 104.28.1.1 (dst port 32451) UDP, length 1400
- IP 5.196.85.159 (src port 27960) ¿ 104.28.1.1 (dst port 32451) UDP, length 978

2. Can you identify the attacker's IP address? And the victim's one? Why?

3. As the network security administrator for the network hit by this attack (i.e., you control the border firewall and can add arbitrary rules), can you mitigate the effect of this attack or prevent it altogether? Why?

4. Consider the following mitigation implemented by the Quake III Arena server: when an IP address sends a getstatus command, the server will check if sender IP address has exceeded a pre-defined rate limit of 10 commands in a period of one second; if the rate limit is exceeded, the IP address is banned from the server forever. I s this solution effective to mitigate the impact of the DoS scenario in the above attack? Why?

5. As the author(s) of Quake III Arena, you want to change the protocol to remove completely the issue that allows it from being exploited for DoS attacks. How would you change the protocol to achieve this goal?

Solution

1. The protocol allows an amplification-based denial of service: in fact, it is a UDP-based protocol, where a request of length 56 triggers a response of 1373 (in the example above), leading to a bandwidth amplification factor (BAF) of 24, i.e., amplifying the attacker's bandwidth of a factor of 24 (which means that, extremely roughly, if the attacker has a 100 Mbps network, given enough open Quake III servers with enough bandwidth each, it can flood the victim with a 2400 Mbps traffic). Concretely, the attacker (given he is in a network that allows IP spoofing) looks on the Internet for N open Quake III servers. It spoofs the victim's IP address, and sends to such servers M UDP getstatus packets. Each server will reply, for each packet received, with with the (long) information packet but, as the source IP is spoofed, the M * N packets will go to the victim, instead of the attacker

2. Attacker: no, the IP address you see in the logs are the ones of the vulnerable Quake III servers, not of the victim. Also, the vulnerable Quake servers can't identify the attacker's IP address as they're spoofed with the victim's IP. Victim: yes, it's 104.28.1.1 (actually it could also be the border firewall or, in general, the company's network).

3. No. Due of the amplifying properties of the protocol, if the attacker has enough bandwidth that, amplified by the 24x factor of the protocol, is greater or equal than the victim's bandwidth, the attacker always succeeds. However, if in this specific scenario the victim is the specific machine 104.28.1.1 and the bottleneck is not given by the network bandwidth of the Internet so company link, but from something else (e.g., the bandwidth of an internal network link, the capabilities of the machine itself, the capabilities of some network middlebox) implementing a firewall rule to drop UDP packet from source port 27960 can mitigate the attack (and thus it would be a good idea to implement). In general, though, a more powerful DoS attack may be launched to defeat this mitigation.

4. No. While this mitigation restricts an attacker to exploit a vulnerable server as an amplifier at most 10 packets per second (i.e., 13730 bytes/s, i.e., about 100 kbps), given that enough vulnerable servers are available the attacker can just use multiple vulnerable Quake III servers at once to defeat the rate limit. Moreover, if the attacker is targeting a network and not a specific IP address, it can spoof multiple IP of the same network, defeating the rate limiting.

5. Solution 1: Implement an handshake at the UDP protocol level (making sure it does not have amplifying capabilities). E.g., the client sends getinfo, the server responds with a nonce, and the client sends the nonce back to the server, then the server sends the long information message. Solution 2: Move the protocol to TCP instead of UDP as a transport protocol. Due to the three-way handshake, TCP is immune to the amplification issue.

CHAPTER 6

Malware

6.1 Exercise 1

A new malware just broke out, causing a world-wide infection and a huge amount of damages. Unfortunately, all the anti-malware systems are not able to detect this malware. You were able to retrieve a couple of samples. Consider the code snippets reported below, extracted from the two malware samples you retrieved:

```
pop ebx
lea ecx, [ebx + 42h]
push ecx
push eax
push eax
sdt [esp - 02h]
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

```
pop ebx
lea ecx, [ebx + 42h]
push ecx
push eax
nop
push eax
inc eax
sdt [esp - 02h]
dec eax
pop ebx
add ebx, 1Ch
cli
mov ebp, [ebx]
```

1. It is clear that the malware is showing evasive behavior. What technique is implemented?
2.

Solution

1. If there is no packing like in this case we have metamorphism. The idea of this malware is to add useless instruction to change the dimension of the code but not the way it works.

6.2 Exercise 2

A new malware just broke out, causing a world-wide infection and a huge amount of damages. Unfortunately, all the anti-malware systems are not able to detect this malware. You were able to retrieve a couple of samples. Consider the code snippets reported below, extracted from the two malware samples you retrieved:

```
0000000000000675 <decrypt>:
[...]
    6a3: 83 f1 42 xor ecx,0x42
[...]
00000000000007b0 <payload>:
    7b0: 28 00 sub BYTE PTR
[rax],al
    7b2: 1a bc 86 0a db 10 0a sbb bh,BYTE PTR
[rsi+rax*4+0xa10db0a]
    7b9: fd std
    7ba: 6d ins DWORD PTR
es:[rdi],dx
    7bb: 20 2b and BYTE PTR
[rbx],ch
    7bd: 2c 6d sub al,0x6d
    7bf: 6d ins DWORD PTR
es:[rdi],dx
    7c0: 31 2a xor DWORD PTR
[rdx],ebp
    7c2: 15 16 1c 0b cb adc eax,0xcb0b1c16
    7c7: 92 xchg edx,eax
    7c8: 0b cb or ecx,ebx
    7ca: 90 nop
    7cb: 4d rex.WRB
    7cc: 47 rex.RXB

0000000000000675 <decrypt>:
[...]
    6a3: 83 f1 42 xor ecx,0x12
[...]
00000000000007b0 <payload>:
    7b0: 78 50 js 802
<__GNU_EH_FRAME_HDR+0x32>
    7b2: 4a ec rex.WX in al,dx
    7b4: d6 (bad)
    7b5: 5a pop rdx
    7b6: 8b 40 5a mov eax,DWORD PTR
```

```
[rax+0x5a]
    7b9: ad lods eax,DWORD PTR
ds:[rsi]
    7ba: 3d 70 7b 7c 3d cmp eax,0x3d7c7b70
    7bf: 3d 61 7a 45 46 cmp eax,0x46457a61
    7c4: 4c 5b rex.WR pop rbx
    7c6: 9b fwait
    7c7: c2 5b 9b ret 0x9b5b
    7ca: c0 .byte 0xc0
    7cb: 1d .byte 0x1d
    7cc: 17 (bad)
```

1. It is clear that the malware is showing evasive behavior. What technique is implemented? How this technique works?

In order to avoid signature detection, a malware sample saves his own assembly code in text format on the victim machine, and then uses a standard assembler to generate and execute the real malicious object code on the machine.

2. How can a signature-based detection method (e.g., antivirus) detect this kind of malware ?

3. You suspect that your machine have been compromised with a kernel rootkit. You tried to use network traffic tools from your machine but you do not see any malicious traffic. Can you conclude that your machine is safe? If is not there are other way to prove you have been compromised?

4. A colleague suggests to replace the hard drive of a machine to be sure to get rid of a very sophisticated rootkit. However, after reinstalling the operating system, it seems like that the machine is infected by the same rootkit. Provide an explanation of what happened. Whatever your answer is, explain why.

Solution

1. Polymorphism.

2. Have a signature to detect the malware in its textual assembly format. Note that having a signature to detect the assembler is a wrong solution, as it leads to lots of false positives (the system assembler it's a legitimate program, after all).

3. No you cannot conclude that the machine have not been compromised. Because the malware can hide its own traffic from tools running on the compromised machine. You could inspect network traffic using an external machine as a MitM between your machine and the router.

4. If it is a BIOS rootkit then no. If it is a kernel rootkit it is ok to just replace the HD or even just reinstall the OS.

6.3 Exercise 3

Our systems have been compromised by a very powerful malware that encrypted all the files in the /home directories of our systems. Luckily, we succeeded in collecting two samples. Luckily, we succeeded in collecting two versions of the same malware. The code of the two malware samples is reported below.

```
.text
0x08048047<evil>:
0x08048047: push ebp
0x08048049: mov ebp,esp
0x08048050: mov eax,0x8090000
0x08048055: mov ebx,0x8048086
0x0804805a: mov cl,BYTE PTR [eax]
0x0804806c: mov dl,BYTE PTR [ebx]
0x0804806e: cmp cl,0x0
0x08048071: je 0x08048086
0x08048077: xor dl,cl
0x08048079: mov BYTE PTR [ebx],dl
0x0804807b: add eax,0x1
0x0804807e: add ebx,0x1
0x08048081: jmp 0x0804805a
0x08048086: sub DWORD PTR [ebx+0x2e],eax
0x08048089: push DWORD PTR [edi]
0x0804808b: sub ebx,DWORD PTR [eax-0x3f66bf99]
0x08048091: retf
0x08048092: mov bh,0xdf
0x08048094: sub ebp,DWORD PTR [ebx+0x3ef5045b]
0x0804809a: imul ecx,DWORD PTR [eax+0x40],0x31
0x0804809e: and al,0x39
0x080480a0: pop eax
0x080480a1: mov edi,0x5d97e942
0x080480a6: loope 0xffffffb2
0x080480a8: cli
0x080480a9: (bad)
0x080480aa: (bad)
0x080480ab: movs DWORD PTR es:[edi],DWORD PTR ds:[esi]
0x080480ac: shl BYTE PTR [ecx],1
0x080480ae: leave
0x080480af: ret
.rodata
0x08090000: <Data with length 40, null terminated>
```

```
.text
0x08048046<evil>:
0x08048046: push ebp
0x08048048: mov ebp,esp
0x0804804a: mov ebx,0x0804806b
0x0804804f: xor eax,eax
```

```
0x08048051: cmp eax,0x28
0x08048054: jge 0x0804806b
0x0804805a: mov cl,BYTE PTR [ebx + eax]
0x0804805d: sub cl,0x3
0x08048060: mov BYTE PTR [ebx + eax],cl
0x08048063: add eax,1
0x08048066: jmp 0x08048051
0x0804806b: fstp QWORD PTR [eax+eiz*2+0x22e70b6d]
0x08048072: jg 0xfffffffa
0x08048074: push ebp
0x08048075: lea ecx,ds:0xf130af40
0x0804807b: fisub WORD PTR [edx-0x703c814d]
0x08048081: mov eax,edi
0x08048083: or eax,DWORD PTR ds:0xd19fe443
0x08048089: bswap esp
0x0804808b: stos DWORD PTR es:[edi],eax
0x0804808c: sbb eax,DWORD PTR [eax+0x50]
0x0804808f: fimul WORD PTR [esp+edi*4]
0x08048092: (bad)
0x08048093: leave
0x08048094: ret
```

1. It is clear that the malware is showing evasive behavior. What technique is implemented? Explain how it is implemented in this specific case. If this malware is metamorphic, explain which instructions implement the obfuscation functionality. If this malware is evasive, explain how it is evading the environment. If this malware is polymorphic, describe how the decryption routine is implemented.

2. We executed the two samples in a Virtual Environment. Tracing the system calls of two different executions with strace yields the following outputs (trimmed for readability):

```
...
open(\/home/law/.bashrc", O_RDONLY) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=3979, ...}) = 0
read(3, 0x7f508000, 3979) = 3979
close(3) = 0
open(\/home/law/.bashrc", O_WRONLY|O_TRUNC) = 3
write(3, 0x7f508000, 3984) = 3984
close(3) = 0
...
(similar lines as above)
...
exit(0)
```

```
...
sleep(0x1000)
exit(0)
```

What kind of technique does the malware employ? Is such a technique adopted by both samples? How can you spot it in both cases?

3. Is it possible to analyze this malware through static analysis? Why? Which instructions would you consider for fingerprint based detection? Do you expect this would be effective?

Solution

1. Polymorphism, the two malware samples employ an OTP (1st sample) and a shift-cipher (2nd sample) with $K = 3$ (Caesar's Cipher) to decrypt/unpack the malicious code. Can't say if the malware is evasive or not since some part of the malware is packed.

2. The malware employs an evasive technique called dormant code. Since it detects that it is running in a VM, it doesn't show its malicious behavior. Such a technique is performed with a sleep of 0x1000 seconds. Only the second sample employs dormant code by performing a sleep syscall, clearly visibile in the strace of its execution.

3. No, because we only see the decryption routines. No relevant instructions can be considered for a fingerprint based detection since the ones of the decryption routines are too generic and would lead to many false positives.

6.4 Exercise 4

Last week, we succeeded in blocking a new malware able to cause a world-wide infection and a huge amount of damage. Consider the code snippets reported below, extracted from the malware sample you retrieved:

```
08048454 <main>:
    8048454: 8d 4c 24 04 lea ecx,[esp+0x4]
    8048458: 83 e4 f0 and esp,0xfffffff0
    804845b: ff 71 fc push DWORD PTR [ecx-0x4]
    804845e: 55 push ebp
    804845f: 89 e5 mov ebp,esp
    8048461: 51 push ecx
    8048462: 83 ec 04 sub esp,0x4
    8048465: 83 ec 0c sub esp,0xc
    8048468: 68 00 01 00 00 push 0x1000
    804846d: e8 8e fe ff ff call 8048300 <sleep@plt>
    8048472: 83 c4 10 add esp,0x10
    8048475: e8 c1 ff ff ff call 804843b <evil>
    804847a: b8 00 00 00 00 mov eax,0x0
    804847f: 8b 4d fc mov ecx,DWORD PTR [ebp-0x4]
    8048482: c9 leave
    8048483: 8d 61 fc lea esp,[ecx-0x4]
    8048486: c3 ret
```

1. It is clear that the malware is showing evasive behavior. What technique is implemented? Explain how it works in this specific case.

Today, a new version of the same malware has been released. Unfortunately, all the anti-malware systems are not able to detect it. You were able to retrieve a couple of samples. Consider the code snippets reported below, extracted from the two malware samples you retrieved:

```
08048340 <main>:
    8048340: 8d 4c 24 04 lea ecx,[esp+0x4]
    8048344: 83 e4 f0 and esp,0xfffffff0
    8048347: ff 71 fc push DWORD PTR [ecx-0x4]
    804834a: 55 push ebp
    804834b: 89 e5 mov ebp,esp
    804834d: 51 push ecx
    804834e: 83 ec 10 sub esp,0x10
    8048351: 21 c0 and eax,eax
    8048353: 90 nop
    8048354: 40 inc eax
    8048355: 09 c0 or eax,eax
    8048357: 48 dec eax
    8048358: 6a 01 push 0x1000
    804835a: e8 a1 ff ff ff call 8048300 <sleep@plt>
    804835f: 21 c0 and eax,eax
    8048361: 90 nop
    8048362: 40 inc eax
    8048363: 09 c0 or eax,eax
    8048365: 48 dec eax
    8048366: e8 15 01 00 00 call 8048480 <evil>
    804836b: 21 c0 and eax,eax
    804836d: 90 nop
    804836e: 40 inc eax
    804836f: 09 c0 or eax,eax
    8048371: 48 dec eax
    8048372: 8b 4d fc mov ecx,DWORD PTR [ebp-0x4]
    8048375: 83 c4 10 add esp,0x10
    8048378: 31 c0 xor eax,eax
    804837a: c9 leave
    804837b: 8d 61 fc lea esp,[ecx-0x4]
    804837e: c3 ret

08048443 <main>:
    8048443: 8d 4c 24 04 lea ecx,[esp+0x4]
    8048447: 83 e4 f0 and esp,0xfffffff0
    804844a: ff 71 fc push DWORD PTR [ecx-0x4]
    804844d: 55 push ebp
    804844e: 89 e5 mov ebp,esp
    8048450: 51 push ecx
    8048451: 83 ec 04 sub esp,0x4
    8048454: 68 00 00 01 00 push 0x10000
    8048459: e8 c6 ff ff ff call 8048424 <what_am_i_doing>
    804845e: 83 c4 04 add esp,0x4
```

```
    8048461: e8 a5 ff ff ff call 804840b <evil>
    8048466: b8 00 00 00 00 mov eax,0x0
    804846b: 8b 4d fc mov ecx,DWORD PTR [ebp-0x4]
    804846e: c9 leave
    804846f: 8d 61 fc lea esp,[ecx-0x4]
    8048472: c3 ret
08048424 <what_am_i_doing>:
    8048424: 55 push ebp
    8048425: 89 e5 mov ebp,esp
    8048427: 83 ec 10 sub esp,0x10
    804842a: c7 45 fc 00 00 00 00 mov DWORD PTR [ebp-0x4],0x0
    8048431: eb 05 jmp 8048438 <what_am_i_doing+0x14>
    8048433: 90 nop
    8048434: 83 45 fc 01 add DWORD PTR [ebp-0x4],0x1
    8048438: 8b 45 fc mov eax,DWORD PTR [ebp-0x4]
    804843b: 3b 45 08 cmp eax,DWORD PTR [ebp+0x8]
    804843e: 7c f3 jl 8048433 <what_am_i_doing+0xf>
    8048440: 90 nop
    8048441: c9 leave
    8048442: c3 ret
```

2. It is clear that the malware is showing evasive behavior. What technique/s is/are implemented? Explain how it works in these specific cases.

3. Consider now malware analysis techniques. Briefly discuss which is the best technique to detect the malware behavior implemented in the previous samples. Motivate your answer by also providing the information you are able to extract in the case of this specific malware.

Solution

1. Dormant code and then a call to the evil function is performed.

2. Metamorphism to evade static analysis. 1st snippet: useless instructions added (nop; and eax, eax;). 2nd snippet: sleep substituted by a cycle that does not do anything (nop).

3. Static analysis for the dormant code; the function sleep@plt easy to spot through static analysis. It is harder to spot loops that act as sleeps. No signature can be extracted. Perhaps, detection of loops that iterates many times. Dynamic analysis can be tried. For instance, waiting for malware behavior over a fixed period of time.

6.5 Exercise 5

Tomb-Bino is a novel malware family that spreads through email attachments, turning infected computers into peers of a command and control infrastructure. An attacker can then command an infected computer remotely, for instance, to perform DDoS, steal data, etc. Its body comprises three functions, which are loaded into memory when the victim clicks on the email attachment: spread, mutate, and sit_and_wait. spread performs the following operations (in order):

1. Steals the list of the user's contacts.
2. Calls mutate.
3. Creates a new file, new_attachment.jpeg.exe, whose instructions are dumped from the process memory.
4. Sends new_attachment.jpeg.exe to every user in the contact list as an email attachment.

mutate takes as input (start_address, number_of_bytes) and substitutes each instruction found in memory at [start_address, (start_address + number_of_bytes)] with a semantically equivalent one. For instance, instruction sub eax, 0x1 is substituted with add eax, -0x1. Substitutions are performed directly to the memory. Lastly, sit_and_wait simply waits for commands from the server and executes them. The following is a simplified view of the process memory when a Tomb-Bino instance is executed:

```
0x00100000 <spread>
(1000 bytes long)
0x001003e8 <sit_and_wait>
(2000 bytes long)
0x00100bb8 <mutate>
(4000 bytes long)
```

1. What type of class does this malware most likely belong to (encrypted, polymorphic, metamorphic, evasive)? Motivate your answer.
2. Suppose that the headers (i.e., library imports, file details, etc) of each "new_attachment.jpeg.exe" cannot be used to generate signatures. Moreover, suppose that spread invokes mutate with parameters (0x00100000, 3000). Could this family of malware be detected through signatures? If yes, explain why. If not, propose a different detection strategy.
3. Given the nature of Tomb-Bino (i.e., a command and control trojan that spreads via email), how would you employ dynamic analysis to detect it? In particular, what kind of resources or operations would you monitor? (Note: a description of command and control malware is provided at the beginning of this section)

Solution

1. Metamorphic: mutate produces semantically equivalent but different code.
2. Yes, it can be detected through signatures. When the malware mutates (i.e., when the function mutate is invoked), it only changes the first two functions, leaving the mutation engine (i.e., mutate) unchanged. Therefore, an antivirus could use mutate as a target for its signatures.
3. Email access (retrieve contacts, send emails) and general networking operations (listen on a port, sending/receiving data to specific IPs).