Computer Security Exam

Professors M. Carminati & S. Zanero

Milan, 02/07/2018

Last (family) Name ______________________________

First (given) Name ______________________________

Matricola or Codice Persona ______________________________

Have you done any challenges/homework, even partially? [ ] Yes [ ] No

Professor [ ] Carminati [ ] Zanero

Instructions

- The exam is composed of 13 pages. Check that you have all of them
- Just as a cross check, tell us whether you have completed the homeworks, by putting an "X" mark appropriately.
- The exam is "closed books". Please put away in a non-suspicious place (i.e. not below the desk) any notebook, or similar. You will be expelled if, at any time, if you do not follow this rule.
- You are not allowed to communicate with other students, and you will be expelled from the exam if you do.
- Shut down and store electronic devices. They will be subject to inspection if found and you may be expelled if you are found using one.
- Please answer within the allowed space. Schemes are good, short answers are recommended.
- You can write in pen or pencil, any color, but avoid writing in red.
- No extra paper is allowed.
- The answers should be written exclusively in the space provided below the questions.

READ CAREFULLY ALL THE POINTS OF EACH QUESTION BEFORE WRITING YOUR ANSWER

SOLUTION

Answer provided in this solution MUST BE CONSIDERED ONLY AS A HINT for the correct answer, and they are not necessarily complete.

Question 1 (10 points)

Consider the C program below.

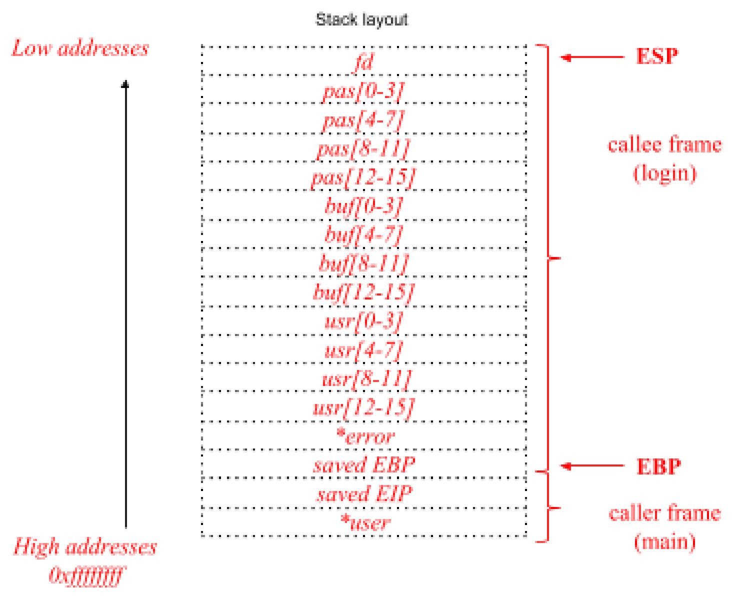
```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <fcntl.h>
#include <unistd.h>

int login(char *user) {
    struct {
        int fd;
        char pass[16];
        char buf[16];
        char usr[16]
        char *error;
    } s;

    s.error = "Access Denied!";
    s.fd = open("pass.txt", O_RDONLY);
    read(s.fd, s.pass, 16); /* load password into memory */
    s.pass[15] = '\0';

    snprintf(s.usr, 16, user);
    printf("Welcome user: %s \n", s.usr);
    scanf("%s", s.buf);
    if(s.buf[0] == '!') {
        if(!strcmp(s.buf, s.pass)) {
            return 1; /* Access OK */
        }
        return 0; /* Access KO */
    } else {
        printf("%s \n", s.error);
        exit(0);
    }
}

int main(int argc, char** argv) {
    login(argv[1]);
}
```

1. [2 points]. 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 17, i.e., at the beginning of the function `login()`, showing:

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

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. [2 points] The program is affected by a typical buffer overflow and a format string vulnerability. Complete the following table, focusing on a vulnerability per row.

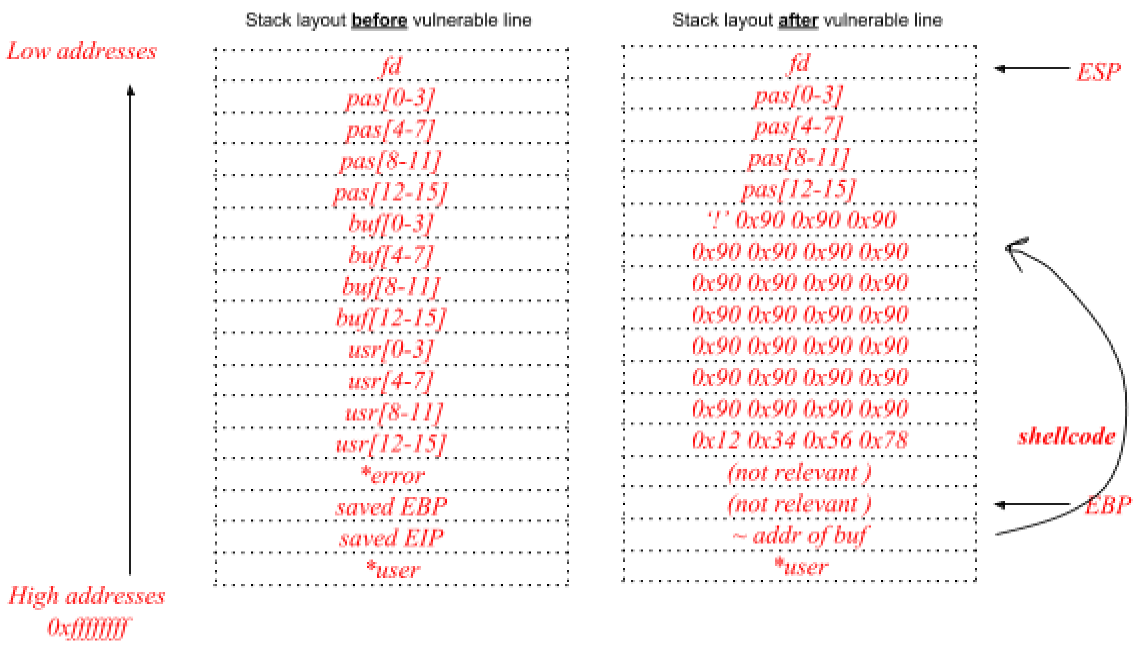
Vulnerability	**Line**	**Motivation**
Buffer Overflow	*Line 24*	*the scanf reads an user-supplied string of arbitrary length and copies it in a stack buffer*
Format String	*Line 22*	*snprintf(s.usr, 16, user) where the format string, user, is directly supplied by the (untrusted) user from CLI argument.*

3. [4 points] 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 this vulnerability to extract the content of the file pass.txt. For this purpose, assume that the following shellcode, composed by 4 bytes of instructions, will output the content of secret.txt: `0x12 0x34 0x56 0x78`.

Write the exploit clearly, detail all the steps and assumptions you need for a successful exploitation, and draw the stack layout right before and after the execution of the detected vulnerable line during the program exploitation.

Command line argument (argv[1]): *any string is OK*



3. [2 points] This time focus on the format string vulnerability you identified. We know the address of the array `s.pass`, which contains the contents of `pass.txt`, and we want to exploit this vulnerability to overwrite the pointer `s.error` with the address of this array.

Assuming we know that:

- The address of the array s.pass (i.e., **what to write**) is 0x42414543 `(0x4241(hex) = 16961(dec), 0x4543(hex) = 17731(dec)`
- The target address (i.e., the address **where we want to write** the address of s.pass) is 0x42414513
- The **displacement on the stack** of your format string is equal to 6

write an exploit for the format string vulnerability to disclose the content of the file pass.txt.

Write the exploit clearly, detailing all the components of the format string, and detailing all the steps that lead to a successful exploitation.

The command line argument is directly fed as a format string to snprintf, and the format string itself is on the stack with a given displacement: we just need to construct a standard format string exploit.

- We need to write 0x42414543 (<tgt>) to 0x42414513
- 0x4241 < 0x4543 -> we write 0x4241 first

The value of the command line argument will have the form

<tgt+2><tgt>%<N1>c%<pos>$hn%<N2>c%<pos+1>$hn

where

<tgt+2> = 0x42414515
<tgt> = 0x42414513
<pos> = 6
<N1> = dec(0x4241) - 8 (bytes already written) = 16961 - 8 = 16953
<N2> = dec(0x4543) - 16953 (bytes already written) = 17731 - 16961 = 770

thus the final string is

\x15\x45\x41\x42%\x13\x45\x41\x4216953c%6$hn%778c%7$hn

Question 2 (10 points)

“WeMessage” is a web messaging application where logged in users can exchange messages containing text as well as basic HTML formatting (such as <b>bold</b> or <i>italic</i>). Furthermore, the web application includes a functionality to manage an user’s personal contact list.

Consider the following code snippets:

Snippet 1: Display a single message - https://chat.example.com/msg?id=<message_id>

```
def display_message(request):
    user = check_session(request.cookies['session'])
    if user is None or request.method != "GET":
        abort(403)
    q = prepared_stmt("select * from messages where id = :id and inbox_user = :user")
    msg = query(q, request.id, user.username)
    if msg is None:
       abort(403)
    page = "<h1>From: " + htmlentities(msg.from) + "</h1>"
    page = page + "<div>" + msg.body + "</div>"
    return render(page)
```

Snippet 2: Send a message - https://chat.example.com/send

```
def send(request):
    user = check_session(request.cookies['session'])
    if user is None:
        abort(403)
    if request.method == "POST":
        q = prepared_stmt("insert into messages (from, to, body)
                           values (:from, :to, :body)")
        if query(q, user.username, request.to, request.text) == True:
            return "<p>Message sent!</p>"
        return "<p>Error sending message</p>"
    else if request.method == "GET":
        return """
                 <form method="POST" action="/send">
                     <p>To: <input type="text" name="to"></input></p>
                     <textarea name="body">Your message...</textarea>
                     <button type="submit">Send</button>
                 </form>
               """
```

Assume the following:

- the function `check_session()` securely manages the users’ sessions
- the function `htmlentities()` converts special characters such as `<`, `>`, `"`, and `'` to their equivalent HTML entities (i.e., `&lt;`, `&gt;`, `&quot;` and `&apos;` respectively)

1. [3 points] Considering only the information given in the previous page, identify which of the following common classes of web vulnerabilities are for sure present in the "WeMessage" application.

Vulnerability class	*Is there a vulnerability belonging to this class in the code? If so, explain why it is present and specify how an adversary could exploit it.*	***If the vulnerability is present in the code above**, explain the simplest procedure to remove this vulnerability while preserving the intended functionalities of the page.*
Cross-site scripting (XSS)	(please also specify which subclasses of XSS are present, e.g., stored\reflected\DOM-based) *Yes, there is a stored XSS vulnerability in the body of the displayed chat messages. An adversary can exploit it by sending to a contact a malicious chat message containing arbitrary Javascript code (e.g.., <script>...</script>).*	*The simplest solution would be to wrap the display of the message body with htmlentities(), i.e., htmlentities(msg.body), as the code already does for the sender. However, this solution does not fulfill the requirement of displaying limited HTML markup, thus it doesn't entirely preserve the intended functionalities of the page. To tackle this, we could, e.g., create a whitelist (not a blacklist!) of non-malicious tags, e.g., <b></b>, <i></i> as well as allowed characters, run the whitelist against the message body and subsequently deleting or escaping any other special character that is not forming a whitelisted tag. Alternatively, we can use a HTML parser and remove (or render as text, i.e., converting special characters to HTML entities) any node or attribute not in the whitelist.*
SQL injection	*No (the application uses prepared statements)*	
Cross site request forgery (CSRF)	*Yes, there is a CSRF in the message sending functionalities. Adversaries can send email messages to logged in users, e.g., by luring them into opening links to their websites (hosted wherever the attackers want) that auto-submit a form to https://chat.example.com/send. The form will be submitted with the correct user's cookies, and a message will be sent on the user's behalf.*	*Implement an anti-CSRF token, e.g., as an hidden input field in the send message form (see slides for details of this technique).*

2. [2 points] Many web application security vulnerabilities stem from the fact that untrusted data is improperly mixed with control information (“code”).

- Describe the three basic validation techniques (blacklisting, whitelisting and escaping) explaining the differences between them, their advantages\disadvantages
- How does the order of application of the three techniques influence their effectiveness?
- What mechanism is the `htmlentities()` function of the above code an instance of? Why did the developers choose to implement this technique in this context?

1) *See slides*
2) *See slides*
3) *htmlentities() is an escaping function. Users may need to include characters such as <, >, ‘, “ in the sender field, and such characters have a special meaning in the HTML context: we can’t just whitelist them (as the site would be vulnerable), and building a blacklist of unsafe sequences\tags is not effective (see slides for why). Thus, we need to escape them.*

3. [1 point] In the context of computer malware, briefly describe in general what is a worm, and what is its most important feature.

See slides on malicious software. The most important feature is their ability to self-propagate.

4. [2 point] One of the vulnerabilities you found in WeMessage can be used to implement a worm-like behaviour (based on web vulnerabilities…), triggered when a user opens a malicious chat message. Please state which of the vulnerabilities can be exploited for this purpose, and describe how the vulnerability can be used to implement the worm.

Vulnerability Exploited: XSS

Malicious behavior: *the attacker sends a chat message with a malicious Javascript script that contains the worm. Thanks to the XSS vulnerability, when the recipient opens the message, the script is executed. Besides performing its malicious activities, the script can easily read the recipient’s contact list (same origin + user is authenticated) and send a message, appearing to come from the original recipient of the worm…., to each contact with the same script (the worm) in the message body. When each contact opens the message, the worm executes again and spreads again to all the contacts.*

5. [2 points] WeMessage implemented a mitigation for a specific vulnerability: the application includes, in each HTTP response from each page served from their domain, the following additional HTTP header:

```
Content-Security-Policy: "script-src 'self'; img-src *"
```

5.1 Briefly explain, in general, what is the purpose of a Content Security Policy (CSP) header.

See slides.

The CSP is meant to limit the provenance of the resources embedded in a webpage, such as scripts, fonts, images or stylesheet, form targets, ...; It is mainly used as a mitigation against XSS (or similar) vulnerabilities as it can define some scripts or origins as trusted or not for serving scripts.

5.2 Which of the vulnerabilities you identified in the WeMessage application is the above CSP header supposed to mitigate? Is this implementation effective in this specific context? (if you think the mitigation is effective, explain why; otherwise, please shortly describe a way to bypass it)

The above CSP header is supposed to mitigate the stored XSS in WeMessage, by restricting the origins allowed to serve trusted Javascript code.

If we assume that there is no vulnerable Javascript loaded in the page or present on the server, this mitigation is effective -- a <script></script> (inline script) injection would be blocked by the CSP as the unsafe-inline directive is not present.

5.3 Consider that WeMessage implements an additional functionality: users can upload *arbitrary attachments* alongside their messages. All the content (including users' message and their attachments) is served from the origin https://chat.example.com, and no vulnerability was fixed while implementing this (except the introduction of the CSP header).
Does your answer to the previous question change? Why?

Yes, the mitigation is not effective.

Due to attachments, users can upload malicious JS attachments to the server and include them with <script src="https://chat.example.com/path/to/the/attachment"></script>. The script, in this case, is not an inline script and is served from the same origin of the page, thus it is allowed by the directive script-src 'self' and executed.

Question 3 (7 points)

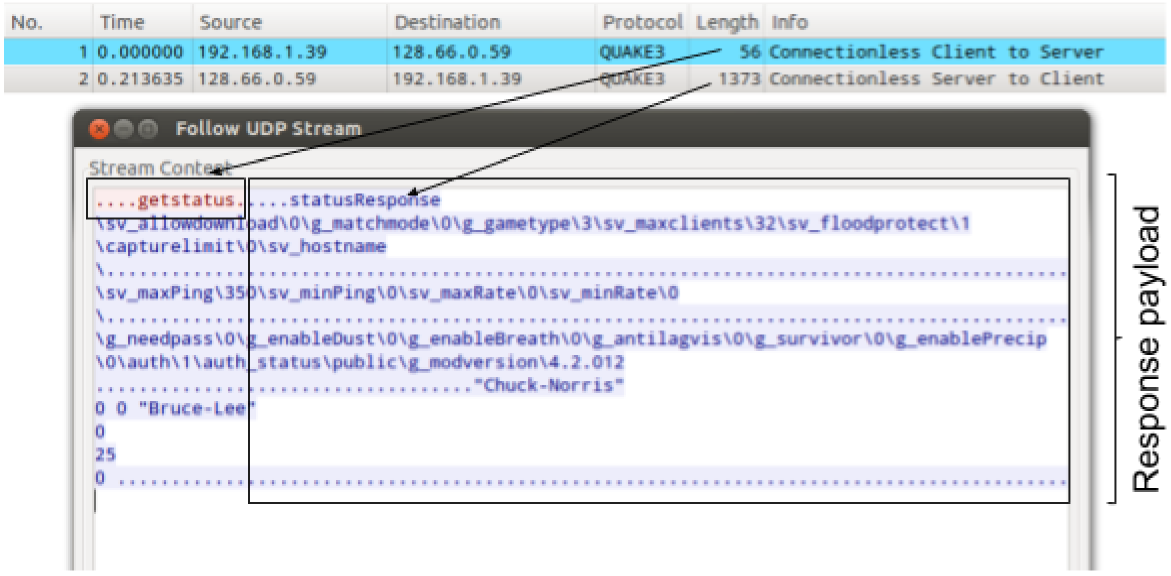
1. [1 point] What is, in general, the goal of a DoS attack? What are the differences between a DoS attack and a distributed DDoS attack? Is, given enough resources, a DDoS attack always feasible? Why?

See slides.

Now consider the "Quake III Arena Network Protocol", a stateless client-server protocol used by the classic multi-player game. Quake III supports multiple game 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.



No.	Time	Source	Destination	Protocol	Length	Info
1	0.000000	192.168.1.39	128.66.0.59	QUAKE3	56	Connectionless Client to Server
2	0.213635	128.66.0.59	192.168.1.39	QUAKE3	1373	Connectionless Server to Client

```
....getstatus.....statusResponse
\sv_allowdownload\0\g_matchmode\0\g_gametype\3\sv_maxclients\32\sv_floodprotect\1
\capturelimit\0\sv_hostname
\..........................................................................................
\sv_maxPing\350\sv_minPing\0\sv_maxRate\0\sv_minRate\0
\..........................................................................................
\g_needpass\0\g_enableDust\0\g_enableBreath\0\g_antilagvis\0\g_survivor\0\g_enablePrecip
\0\auth\1\auth_status\public\g_modversion\4.2.012
......................................"Chuck-Norris"
0 0 "Bruce-Lee"
0
25
0 .......................................................................................
```

2. [2 points] 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.

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\she 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.*

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
```

3. [1 point] Can you identify the attacker's IP address? And the victim's one? Why?

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)

4. [1 point] 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?

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 >= 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 → 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…

5. [1 point] 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*".
Is this solution effective to mitigate the impact of the DoS scenario in the above attack? Why?

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.

6. [1 point] 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: 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.

Question 4 (7 points)

1. [3 points] What is an antivirus? Which are the strategies through which it detects malware? State their names and give a brief explanation of how they work.

See slides.

2. [2 points] What are the techniques used by malware to evade detection? State their names and give a brief explanation of how they work.

See slides.

3. [2 points] Consider mobile malware developed for the Android platform. Can an antivirus for Android be as effective as an antivirus developed for a legacy operating system (e.g., Windows, or Linux) in the task of detecting a malware already running in the system? Why?

See slides.