Computer Security Exam

Professors F. Maggi & S. Zanero

Milan, 04/09/2017

Last (family) Name _____

First (given) Name _____

Matricola or Codice Persona _____

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

Professor [ ] Maggi [ ] Zanero

Instructions

- The exam is composed of 12 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 note, book, 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.

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 (8 points)

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

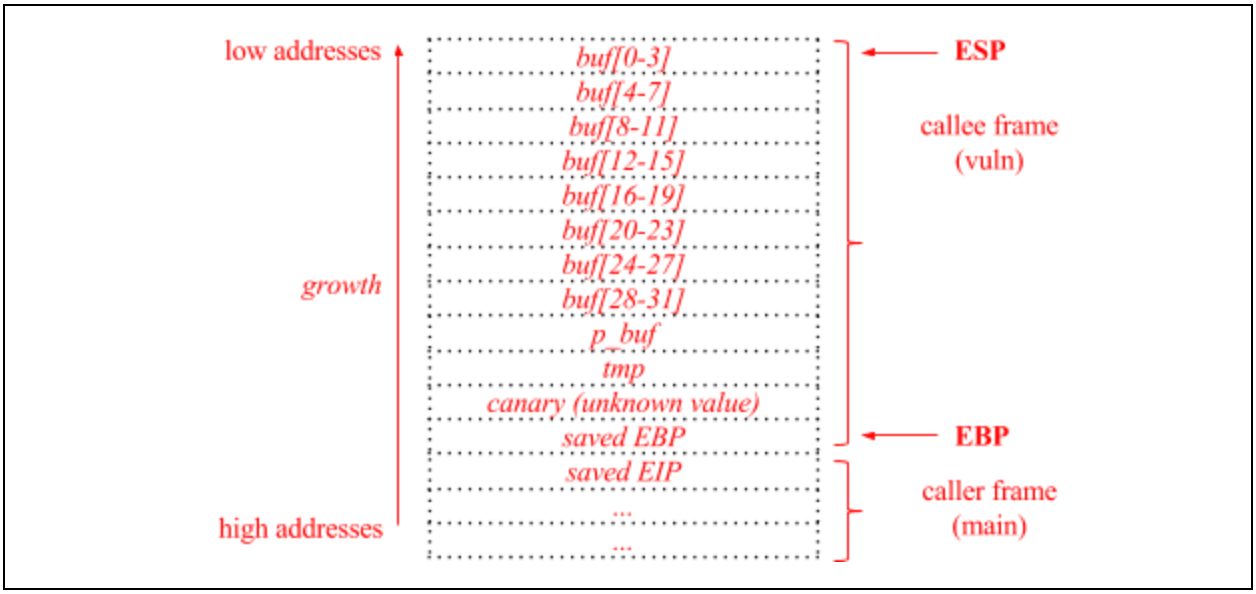
```
1   #include <stdio.h>
2   #include <stdlib.h>
3   #include <string.h>
4
5   void vuln() {
6     struct {
7         char buf[32];
8         char *p_buf;
8         int tmp;
9     } locals;
10
11    locals.p_buf = locals.buf;     //  <--------- here
12    scanf("%s", locals.p_buf);
13    if (strncmp(locals.buf, "Knight_King!", 12) == 0) {
14       printf("location: %x", locals.p_buf);
15       scanf("%s", locals.p_buf);
16       printf("message: %s", locals.p_buf);
17    }
18  }
19
20  int main(int argc, char** argv) {
21    vuln();
22  }
```

1. [2 points] Assume the usual IA-32 architecture (32-bits), with the usual "`cdecl`" calling convention. Also assume that the program is compiled using stack canaries as the only mitigation against exploitation (the address space layout is not randomized, and the stack is executable).

Draw the stack layout when the program is executing the instruction at line 11, showing:

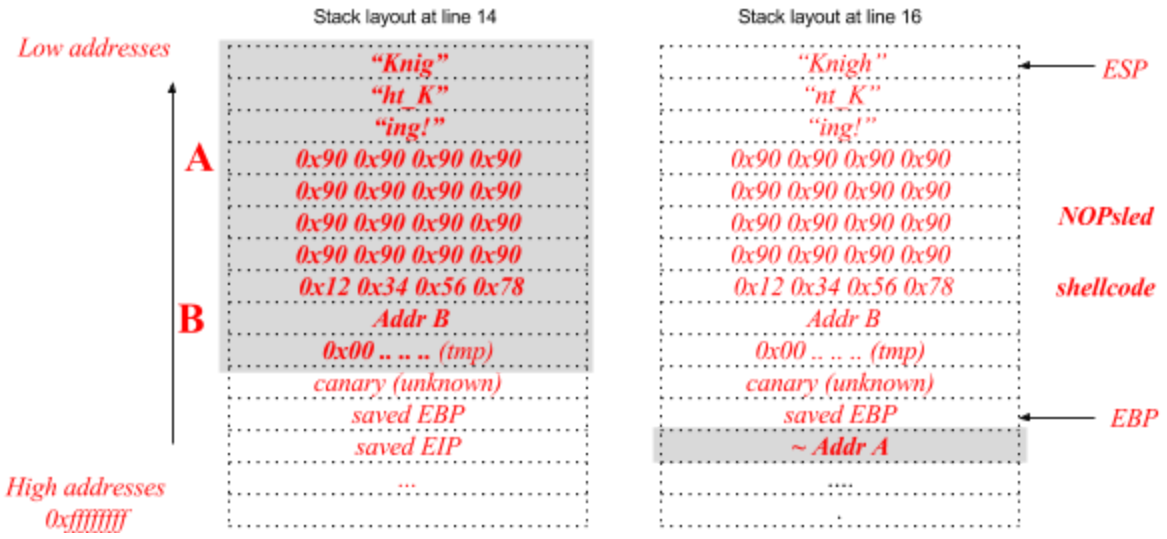
a. Direction of growth and high-low addresses;
b. The name of each allocated variable;
c. The boundaries of frame of the function frames (`main` and `vuln`).

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. [4 points] Write an exploit for the buffer overflow vulnerability in the above program. Your exploit should execute the following simple shellcode, composed only by 4 instructions (4 bytes): `0x12 0x34 0x56 0x78`. Write clearly all the steps and assumptions you need for a successful exploitation, and show the stack layout after the execution of each `scanf()` during the program exploitation.

- *Step 1: Through the scanf() on line 12, write the shellcode on the stack and overwrite p_buf with the address of a memory location below the canary - namely, the address of the saved EIP (we can assume it's precisely known -- or that we can run the program multiple times with the same environment, and gather the address through the leak of line 14). Also, we need to start the buffer with "Knight_King!" to actually enter the correct branch of the "if" statement.*
- *Step 2: In the scanf() on line 15, we can directly overwrite the saved EIP bypassing the canary.*



3. [1 points] If *address space layout randomization (ASLR)* is active, is the exploit you just wrote still working *without modifications*? Why?
Provided that, this time, the program is compiled without stack canaries, can you describe a way to exploit the program also with ASLR enabled?

- *No, because the **address of the stack** would be **randomized** for every execution, and we must have to have a **leak** in order to exploit it successfully.*
- *If stack canaries are disabled then it would be possible **without overwriting p_buf** to exploit it by using the **leak at line 14**, and then writing the **address of the shellcode** inside the **return address**.*

4. [1 point] If the stack is made non executable (i.e., NX, DEP, or W^X), is the exploit you just wrote still working *without modifications*? If not, propose an alternative solution to exploit the program.

- *No, the exploit executes shellcode from the stack.*
- *We can use ret to **libc** technique in order to execute the **system** function in the libc and obtain a **shell**. In order to do that, we change the value of **B** and let it point to the **address of system**.*

Question 2 (7 points)

Professor Sherlock Holmes is setting up a private wiki to share material about his course on "Advanced Topics in Investigations" with his students. To this purpose, the web administrator of his company assign to every professor a personal website running on the domain homes.nsy.uk. Sherlock's students can access the material using their own credentials at http://homes.nsy.uk/~holmes/wiki.
Professor Moriarty, Sherlock's famous rival, is not allowed to access such material. However, he can publish material to his own website at http://homes.nsy.uk/~moriarty/.

1. [2 points] How can Prof. Moriarty get a copy of a *specific* page from Sherlock's wiki (e.g., http://homes.nsy.uk/~holmes/wiki/Secret)?

a) Professor Moriarty should ask Sherlock or one of his students to visit his page. On his page, he should create an iframe pointing to the secret page on Sherlock's wiki, and read the contents of that frame using Javascript code in her own page. The same-origin policy allows this because both Sherlock's and Moriarty's pages have the same origin (i.e., http://scripts.nsy.uk/).

b) MITM to sniff credentials [+ definition and explanation]

Professor Sherlock gives up on the wiki, and decides to build a system for selling books about his adventure, hosted at http://watsonbook.nsy.uk/ .
The pseudocode for handling requests to http://watsonbook.nsy.uk/buy is as follows:

```
1. def buy_handler(cookie, param):
2.     print "Content-type: text/html\r\n\r\n",
3.
4.     user = check_cookie(cookie)
5.     if user is None:
6.           print "Please log in first"
7.           return
8.
9.     book = param['book']
10.    if in_stock(book):
11.          ship_book(book, user)
12.          print "Order succeeded"
13.    else:
14.          print "Book", book, "is out of stock"
```

where `param` is a dictionary of the query parameters in the HTTP request (i.e., the part of the URL after the question mark), and 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).

2. [3 points] Identify which of the following classes of vulnerabilities are present in Sherlock's code:

Vulnerability class	*Is there a vulnerability of this class in Sherlock's code? Why? How could an adversary exploit it?*	*If the vulnerability is present, explain the simplest procedure to remove it.*
cross-site scripting (XSS)	*Yes, line 14 is vulnerable to cross-site scripting. An attacker can supply a value of book that contained something like <script>alert(document.cookie)</script>, and assuming the in_stock function returned false for that book ID, 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 "book" variable.*

Cross site request forgery (CSRF)	*Yes, an adversary can set up a form that submits a request to buy a book to http://watsonbook.nsy.uk/buy?book=anyid, and 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']. The CSRF stems from the fact that the web application authenticates the user only with "ambient credentials" (cookies, which are sent automatically with every request) also to perform a state-changing action (buy a book in this case); this way, whatsonbook is not able to authenticate whether the request was voluntarily initiated by the user or not. A solution is to require for every state-changing action a further secret token (e.g., automatically inserted as a hidden field in the bank's form, ...) and checking its validity before performing the operation. This way, to buy a book, it is necessary to read the content of a request to whatsonbook's servers, and not just to blindly perform a request. As the same origin policy does not allow a website to read the response of an arbitrary request to a different origin, the attackers would not be able to read the secret CSRF*
Sql Injection (SQL-I)	*No, the code doesn't reference any database. (better yet: we can't say, it depends upon how ship_book and check_cookie are implemented)*	

3. [2 points] As a cybersecurity expert warned Sherlock that it's not a good idea to have an online store over plain HTTP, Sherlock moved his book store to https://www.watsonbooks.com/.
To make his web page interactive, Sherlock needs the popular jQuery Javascript library. He is told that the website will run faster if he loads common assets from a third party CDN (content delivery network). He finds that MyAwesomeProvider, a CDN provider he never heard about before, hosts JQuery in its CDN. Thus, following the provider's tutorial, he adds the following line to his web page:

```
<SCRIPT SRC="http://cdn.myawesomeprovider.com/jquery/3.2.0/jquery.js">
```

Provide two reasons why this is a bad idea from a security perspective and a possible solution (and, in fact, due to one of those two reasons, it may not work in modern web browsers).

Reason 1: *First, it allows an adversary with access to the network of a visitor to Sherlock's web site to inject arbitrary Javascript code into the HTTPS page, bypassing any cryptographic protection Sherlock might have wanted.*

Solution 1: *Load the library from a CDN that supports HTTPS.*

Reason 2: *Second, it allows an adversary that compromises cdn.myawesomeprovider.com (or MyAwesomeProvider itself, if it goes rogue) to serve arbitrary Javascript code to run in browsers that visit Sherlocks's page, even if that adversary doesn't control the visitor's network.*

Solution 2: *Host JQuery on the watsoonbook website itself (even better, but outside the course program: if using an untrusted CDN, use subresource integrity -- SRI, https://developer.mozilla.org/en-US/docs/Web/Security/Subresource_Integrity -- to cryptographically verify the third party resources)*

Question 3 (5 points)

Let's consider **denial of service (DoS) attacks**.

1. [0.5 points] What is, in general, the goal of a DoS attack?

See slides

2. [0.5 points] Describe the difference between a denial of service attack and a *distributed* denial of service attack (DDoS). Why, given enough resources, a DDoS is generally always feasible?

See slides

Consider a diagnostic protocol used to remotely monitor the status of a computer system The protocol works as follows: a server listens over UDP on a well-known port. Upon receiving a "query" message (a packet with a fixed payload of 4 bytes), the server replies with the computer's monitoring information. The size of the UDP response ranges from 500 bytes to 1kB. The protocol is exposed to the whole Internet.

3. [2 points] Please explain how an attacker can misuse the above mentioned protocol to *ease* a DoS attack against a victim (the victim does not need to run this protocol!). Make sure (a) to mention the feature of this protocol that allows the misuse for the purpose of a DoS attack, and (b) to precisely describe the attack scenario.

The protocol allows for an amplification-based DDoS attack, with an amplification factor of up to 256, i.e., the attacker can amplify its bandwidth of a 256 factor because the server replies with 256 times the size of the (unauthenticated) request. The scenario is the following: attacker on network A spoof its IP address with the victim's one, and sends many requests to a server B running this protocol. B will reply to the spoofed IP (i.e., to the victim) with 256 times the data sent to B by the attacker, possibly saturating the victim's bandwidth.

4. [2 points] Consider a variant of this protocol, where the transport is over TCP instead of UDP. Does the TCP-based protocol present the same weakness? Why?

No. The scenario requires that the attacker spoofs its IP address, and the attacker wouldn't be able to complete the initial TCP handshake (modulo vulnerabilities in the TCP implementation, e.g., non-random initial sequence numbers) and trigger the amplification behavior.
The attacker would send a SYN packet to the "amplificator" service, which responds to the victim with a "SYN-ACK". The victim will not reply back with an "ACK" (having not initiated the connection) ~> no amplification takes place.

Question 4 (7 points)

The "Foggy Storage" service lets you store files on the cloud. You can upload, download and view files that you previously sent to the service.

Each user is identified by a login system (username, password) and the connection is securely maintained using session cookies. The application is backed by a SQL database, and all the queries are executed against the following tables:

users

user_id	username	password
1	Aria	af34...
2	John	89ba...
3	Tyrion	56e7...
4	Daenerys	f24c...
5	You	556c...

files

file_id	filename	data
1	unk	...
2	secret.txt	...
3	stuff.jpg	...
4	summer17.jpg	...

permissions

file_id	user_id
1	1
2	1
4	4
3	2

To upload a file, users visit the page *upload.php*:

```
<form method="POST">
   <input type="text" name="filename">
   <input type="file" name="data">
   <button type="submit">Upload!</button>
</form>
```

which is processed by the following PHP-like server-side pseudocode:

```
$filename = $_POST['filename'];
$data = $_POST['file'];
$user_id = $_SESSION['user_id']
// ...

$stmt = $conn->prepare("INSERT INTO files (filename, data) VALUES (?, ?)");
$stmt->bind_param("sb", $filename, $data);
$stmt->execute();

$query = "SELECT file_id, filename FROM files WHERE filename = '" . $filename . "'";
$res =  $conn->execute($query);

echo "id: " . htmlentities($res[0]) . " <br> filename" . htmlentities($res[1]) ;

$stmt = $conn->prepare("INSERT INTO permissions (file_id, user_id) VALUES (?, ?)");
$stmt->bind_param("dd", $res['file_id'], $user_id);
$stmt->execute();
```

Where `htmlentities(string)` converts some characters to HTML entities.

For example:

```
<a href="https://www.example.com">Go to example.com</a>
```

Is converted into:

```
&lt;a href=&quot;https://www.example.com&quot;&gt;Go to example.com&lt;/a&gt;
```

1. [2 points] Identify the class of the vulnerability and briefly explain how it works.

SQL Injection. *See slides for the explanation.*

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***.

2. [2 points] Write an exploit for the vulnerability just identified to get the content of **secret.txt**:

```
' UNION SELECT f.file_id, f.data FROM files AS u WHERE f.name =
'secret.txt'
```

Consider a version of the program where the vulnerability has been fixed:

```
$filename = $_POST['filename'];
$data = $_POST['file'];
$user_id = $_SESSION['user_id'];
// ...

$stmt = $conn->prepare("INSERT INTO files (filename, data) VALUES (?, ?)");
$stmt->bind_param("sb", $filename, $data);
$stmt->execute();

$stmt = $conn->prepare("SELECT file_id, filename FROM files WHERE filename = ?");
$stmt->bind_param("s", $filename);
$res = $stmt->execute();

echo "id: ". htmlentities($res[0]) . " <br> filename" . htmlentities($res[1]) ;

$stmt = $conn->prepare("INSERT INTO permissions (file_id, user_id) VALUES (?, ?)");
$stmt->bind_param("dd", $res['file_id'], $user_id);
$stmt->execute();
```

Also assume that, to download a file, users visit *download.php*, passing the name of the file as a single GET parameter. The page *download.php* is securely implemented, and outputs the file if and only if there is an appropriate permission in the `permissions` table.

3. [2 points] Can you find a way to retrieve the content of **secret.txt** (without the appropriate permissions) in the patched version of the program?
Provide a motivation/explanation supporting your answer.

It is not safe. I can retrieve the secret.txt file by uploading a new file with the same name.

4. [1 point] How would you fix this second vulnerability?

The program lacks even the basic error checking, so the fix for this vulnerability is to add proper error/exception management.
For instance, if 'filename' is supposed to be unique, add a unique constraint in the DB for 'filename', check for this exception in the program and do not update permissions if this exception is thrown. If 'filename' is not supposed to be unique, instead of doing the 'select' query to find the file_id, use the proper DBMS functionalities to get the primary key of the last insertion to avoid updating the permissions of a "wrong" file.

Question 5 [6 points]

You are the network security administrator of a LAN. Despite all your efforts to protect your users, *all the LAN computers are found to be infected by a ransomware more or less at the same time.*
All the infected computers had automatic updates enabled for an accounting software and a new version of the software was automatically downloaded from the Internet over HTTPS and installed right before the infection.

4. [2 point] Can you suspect what is going on (please state explicitly any assumptions you make)?

It is unlikely for an attacker to perform a simultaneous MITM to many different networks (given also the fact that, this time, updates are securely delivered over HTTPS), we can assume that the accounting software's update website was compromised, and that the update package was substituted with a trojanized version containing the ransomware, infecting all the computers in the company.

After this event, you want to prevent this scenario from happening again. The analysis revealed that the ransomware that infected your network was a known malware. The hosts in the network must browse the Web (HTTPS, port 443), but you want to avoid that they download known malware (i.e., malicious programs for which signatures exist).

1. [1 point] Describe the network equipment (if any) necessary to implement it (please state explicitly any assumptions you make).

We need a firewall capable of decoding the application layer in order to trigger on HTTPS responses corresponding to HTTPs requests initiated from the internal network. HTTPs payload is encrypted. Thus, checking it on an AV signature database does not make any sense. In this specific case an HTTPS proxy could be used. Since the firewall needs to decrypt the application-level payload, it must become a MITM during the SSL handshake. To achieve this, we install another trusted CA in the certificate store of each client's browser.

In order to avoid the previous 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. [2 points] How can a signature-based detection method (e.g., antivirus) detect this kind of malware ?

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. [1 point] Imagine you are a malware author. What would you do to avoid signature detection in general? Provide (and explain) two techniques.

Employing polymorphism/metamorphism/obfuscation/packing. The idea is to modify the payload of the malware, so to evade the known signature, without changing its malicious behavior.