Computer Security Exam

Professors F. Maggi & S. Zanero

Milan, 03/07/2017

Last (family) Name _____

First (given) Name _____

Matricola or Codice Persona _____

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

Instructions

- The exam is composed of 10 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.

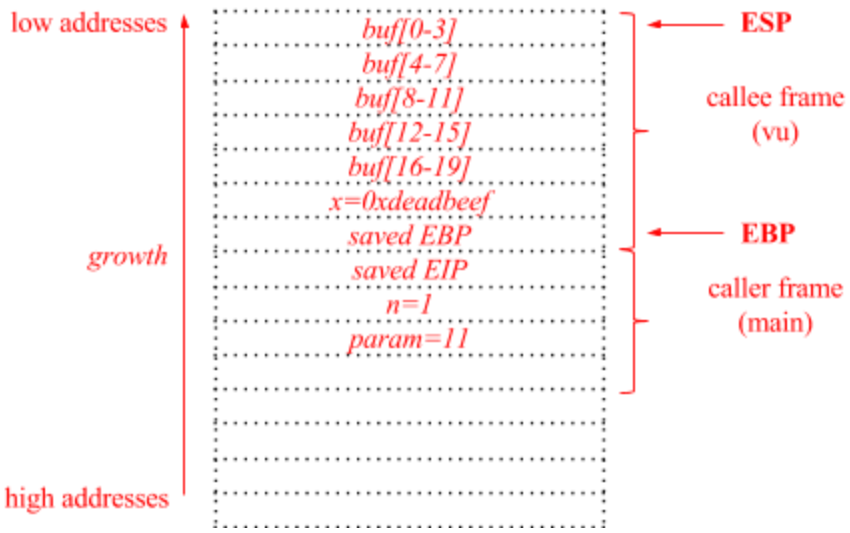
Question 1 (9 points)

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

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

int vuln(int n, int param) {
    int x = 0xdeadbeef;
    char buffer[20];

    if(n < 2 || param > 10) {                        ←--- here
        scanf("%s", buffer);
        if(strcmp(buffer, "sesame") == 0) {
            doSomething(); // not relevant for the exercise
        }
    }

    if(x != 0xdeadbeef) {
       printf("Goodbye.\n");
       abort();
    }
    return 1;
}

int main(int argc, char** argv) {
    vuln(argc, atoi(argv[1]));
    return 0;
}
```

1. [2 points] Assuming the usual IA-32 architecture (32-bits), with the usual `cdecl` calling convention, draw the stack layout at execution point "`←--- here`", showing:
    a. Direction of growth and high-low addresses.
    b. The name and content of each allocated variable.
    c. The boundaries of frame of the function frames (`main` and `func`).

    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. [2 points] The underlined lines of code attempt to mitigate the exploitation of the buffer overflow vulnerability.
    a. Shortly describe what is the implemented technique, and how it works in general.

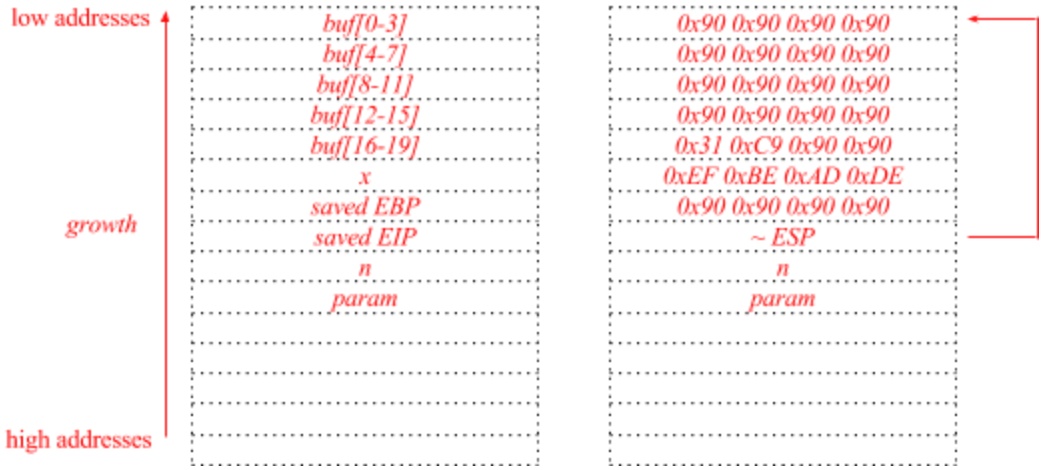
The underlined code implements a stack canary as a mitigation against stack-based buffer overflows. When writing past the end of a stack-allocated buffer, one would overwrite the variable x before overwriting the saved EIP. Before returning from the function, the content of the variable x is checked against the original value: if they differs, a buffer overflow is detected and the program aborts without returning from the function (i.e., without triggering the exploit).

    b. Describe the weaknesses in this specific implementation, and how you would fix them.

In this case, the stack canary is a static value (0xdeadbeef): if the binary is available, it is enough to retrieve this value by reverse engineering the program; then, during the exploitation it is enough to overwrite the stack canary with this (known) value. To fix this issue, the stack canary should be randomized at the program startup and placed in a register.

3. [2 points] Write an exploit for this vulnerability to execute the following simple shellcode, composed only by 2 instructions (2 bytes): `0x31 0xC9`. Make sure that you show how the exploit will appear in the process memory with respect to the stack layout of the previous point. Ensure you include all of the steps of the exploit (i.e., ensuring that the program and the exploit execute successfully).

We have to fill the buffer with: the NOPsled, followed by our shellcode, followed by 0xdeadbeef (to circumvent the broken stack canary), followed by the approximate address of the ESP (i.e., of the NOPsled):

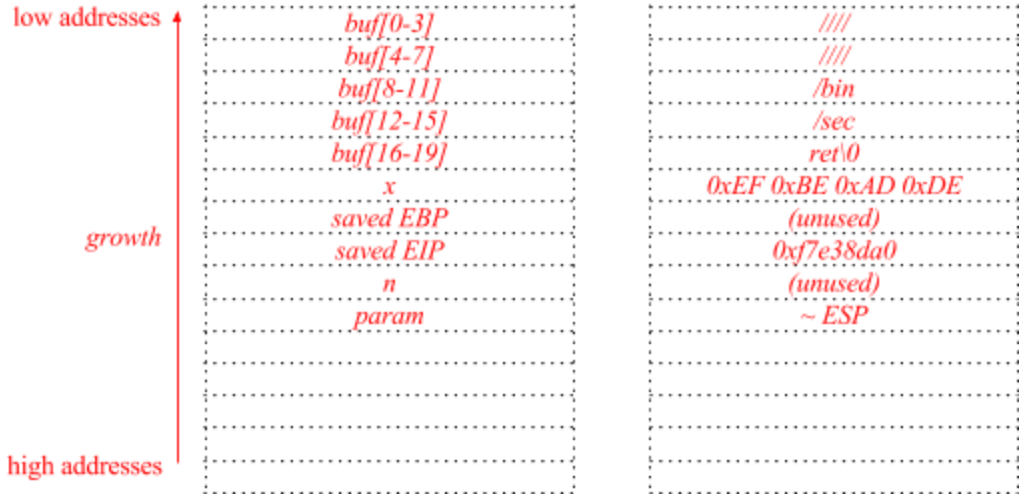


4. [3 points] Assuming that W^X is enabled (i.e., non-executable stack):
    a. Is the exploit you just wrote still working? Why?

No, because we can't jump to our shellcode (the stack is non-executable).

    b. 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`.

We write in the buffer the string /bin/secret, and we overwrite the saved EIP with the address of system(). We then overwrite 'param' with a pointer to ~ ESP so that, when jumping into the system() function, this pointer will be at ESP-4 (i.e., where system() expects the parameter).



    c. Let's now assume that both W^X and ASLR are enabled. Is the exploit developed at point (b.) still working as is? Why?

No, because with ASLR enabled, we don't know neither the address of system() in the libc, nor the address of the string /bin/secret written in the buffer (both the stack and the shared libraries are randomized).

Question 2 (6 points)

You are writing a challenge for a computer security course. You are asked to focus on format string vulnerabilities.

1. [3 points] Write a *vulnerable* program, which is supposed to read password and username as strings from the standard input and authenticate the user.
The program must have exactly one vulnerability, and it must be a basic format string vulnerability. Ensure your code has no other vulnerabilities.
You can skip the code that deals with the authentication: just represent it as a call to a function

```
int authenticate(char* username, char* password);
```

that returns 1 if the credentials are valid, 0 otherwise..

```
int main (int argc, char* argv[]) {

     char username[256];
     char password[256];

     printf("Username: ");
     scanf("%256s", username);
     printf("Password: ");
     scanf("%256s", password);

     if(authenticate(username, password)) {
        printf("Hello, %s\n", username);
     } else {
        printf("Authentication error. Can't authenticate ");
        printf(username);
     }



}
```

2. [3 points] Explain (a) how you implemented a format string vulnerability, and (b) what will happen during exploitation (e.g., what is the logic of the vulnerable program that you wrote).

(a) The instruction printf(username) uses a variable as a format string. The content of the username variable is directly controlled by an attacker.

(b) During exploitation, the attacker will insert as an username a string containing control characters (e.g., %n, %x, …) to basically write an arbitrary value to a memory location, and specify any (invalid) password. For example, the attacker can overwrite the saved EIP of the main() with the address of an attacker-injected shellcode, hijacking the control flow of the program and executing arbitrary code.

Question 3 (6 points)

To protect users from cross-site scripting (XSS) attacks, a browser implements the following procedure:

(a) Before attempting to render a page, the browser scans the URL requested by the user looking for any text that matches the pattern `/<[^>]*>/`*. This regular expression will match an open-tag character (<), followed by an arbitrary number of characters other than a close-tag character, followed by a close-tag character (>);*
(b) if there is a match, the value of the parameter(s) containing the match is marked as dangerous (if the match is not in a parameter, the browser marks as dangerous the portion of the URL starting with the match);
(c) before rendering the response, it scans the received response for any instance of text marked as dangerous in point (b). If any dangerous text is found, the browser will refuse to execute any script contained in the page.

For example, if the user visits one of the following URLs:

```
https://example.com/page?p1=<script>xss()</script>&p2=test
https://example.com/search/<script>xss()</script>
```

and the HTTP response contains the text `<script>xss()</script>`, the browser shows a security warning, and script is not executed.

1. Briefly explain:
    a. [1 points] How **reflected cross-site-scripting vulnerabilities** work

(a) There is a reflected cross-site scripting (XSS) vulnerability when there exist a per-request data flow that originates from the client's rendered page (e.g., input field or other HTTP variable) and ends up on the client's rendered page (e.g., interpreted HTML or Javascript content) as a result of the server response.

b. [2 points] whether, and to what extent, this mechanism protects against **reflected cross-site-scripting vulnerabilities.**

(b) The proposed mechanism prevents the exploitation of reflected XSS vulnerabilities where (a) the "reflected" user-injected data is contained in the URL, (b) the vulnerable application reflects the user's payload in a HTML context, and (c) the response contains the user-injected data exactly as is.
Thus, the proposed mechanism protects only from some exploits against reflected XSS vulnerabilities, not from all of them. For example:

- *If the server processes the URL before reflecting it, then the reflection might not directly correspond to the text inspected by the regular expression. For example, using the normal HTTP hex-escape rules, a server will translate a received URL that includes "...%3cscript%3e..." to "...<script>...". The regular expression will miss matching the original URL, since it does not include literal < and > characters, but the browser will still treat the response as including a <script> tag.*
- *As only the URL is scanned, the procedure will not detect any reflected XSS attacks where the reflected string is in other component of the request (e.g., POST variables, the REFERER header, …).*
- *The procedure will detect attacks against vulnerabilities that reflect user-provided data in the HTML context only. If the user's data is reflected as part of an HTML attribute or in a Javascript script, it is possible to inject a script without < or >, which would not be detected.*

2. Briefly explain
   a. [1 points] how **stored cross-site scripting vulnerabilities** work

In a stored XSS vulnerabilities, there exist a way for a user to store data on the web application that will be retrieved (possibly by other users) at a later time. With respect to reflected XSSes, the victim's request do not contain the malicious payload.

   b. [2 points] whether, and to what degree, this mechanism protects against them.

Thus, the browser has no way to differentiate between a legitimate script and a XSS attack, and this mechanism does not protect at all against stored XSS.

Question 4 (12 points)

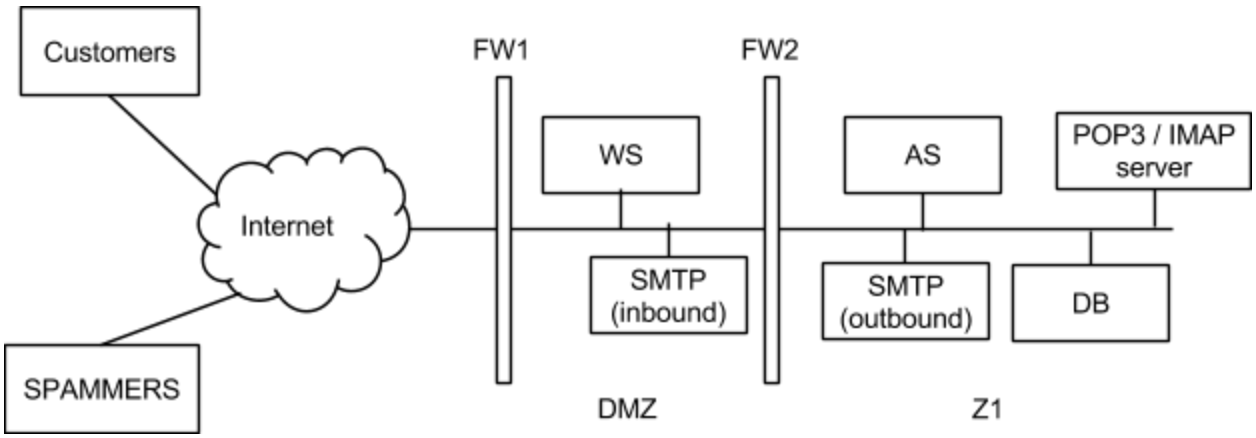
FreeMail is an anti-spam company that aims to fight spam using an innovative "vigilante" approach. FreeMail's customers report their spam e-mails to FreeMail. Then, FreeMail leaves a generic complaint for each spam e-mail reported by users. FreeMail operates on the assumption that, as the community grows, the flow of complaints from hundreds of thousands

of computers will apply enough pressure on spammers and their clients to convince them to stop spamming.

Users can report spam e-mails either through a web application (accessible over HTTPS) or by forwarding the spam messages to a dedicated e-mail address (i.e., inbound e-mails are received by a dedicated SMTP server). The web server uses application logic deployed on an application server. The logic implemented on the application server automatically visits every website advertised by the URLs in the spam messages and leaves complaints on those websites. Complaints are left in the website's contact forms or, if the application logic can't find any contact form, by sending an email to the spammer provider's abuse contact (obtained by querying the WHOIS service). Furthermore, the application server saves in a SQL database information about the spam messages that are reported.

Read **all** the following questions and **then** answer one by one:

1. [1 points] Draw FreeMail's network layout and assign distinct names to any machine and zone.



2. [3 points] Write the firewall rules, assuming firewalls to be stateful packet filters (i.e. you can consider the response rules implicit)

Firewall	Src IP	Src PORT	Direction of the 1st packet	Dst IP	Dst PORT	Policy	Description
FW1 (example)	10.0.0.1 (example)	ANY	zone 1 -> zone 2	192.168.0.2 (example)	443	DENY	(example: the X server in zone 1 cannot contact the Y server)
FW1	*ALL*	*ANY*	*ANY*	*ALL*	*ANY*	*DENY*	*Default deny*

FW1	ANY	ANY	Internet -> DMZ	WS_IP	443 (HTTPS)	ALLOW	The webserver is publicly reachable
FW1	ANY	ANY	Internet -> DMZ	SMTPIN_IP	25	ALLOW	The SMTP server is publicly reachable
FW2	ALL	ANY	ANY	ALL	ANY	DENY	Default deny
FW2	WS_IP	ANY	DMZ → Z1	AS_IP	CUST	ALLOW	The webserver connects to the application server
FW2	SMTPIN_IP	ANY	DMZ → Z1	IMAP_IP	587	ALLOW	SMTPIn relays the incoming e-mails to the POP3\IMAP server (used by the application server)
FW2	AS_IP	ANY	Z1 → DMZ	ANY	80, 443	ALLOW	The application server connects to the spammer's websites
FW1	AS_IP	ANY	DMZ → Internet	ANY	80, 443	ALLOW	The application server connects to the spammer's websites
FW2	SMTPOUT_IP	ANY	Z1 → DMZ	ANY	25	ALLOW	The application server sends email to the abuse contacts (relayed by the SMTPOut server)
FW1	SMTPOUT_IP	ANY	DMZ → Internet	ANY	25	ALLOW	The application server sends email to the abuse contacts (relayed by the SMTPOut server)
FW2	AS_IP	ANY	Z1 → DMZ	ANY	WHOIS	ALLOW	The application server contacts the WHOIS servers
FW1	AS_IP	ANY	DMZ → Internet	ANY	WHOIS	ALLOW	The application server contacts the WHOIS servers

- Dividing Z1 into multiple zones (e.g., to isolate the DB server) is considered acceptable.
- As SMTP_IN is relaying emails only for the application server, it is considered an acceptable solution to have SMTP_IN connect directly to AS to relay messages to it instead of having a separate POP3\IMAP server.
- The last two rules about the WHOIS protocol are optional (i.e., we don't expect everyone to know that WHOIS is a protocol by itself…)

After a short while since the beginning of their operations, FreeMail's public web site comes under a massive **DDoS attack** that uses **SYN flooding**.

3. [2 points] Briefly describe **SYN flooding** attack and how the attack can cause a denial-of-service.

A SYN flooding attack sends a stream of TCP "initial SYN" packets to the targeted server. Each packet appears to represent a request to establish a new connection. An attack that employs a large botnet, for example, might not use spoofing. : For each incoming SYN packet, the server both responds and consumes memory because it records information (state) associated with the impending new connection. The attack primarily aims to exhaust the server's available memory for keeping this state.

4. [2 points] Briefly describe **one countermeasure** that FreeMail could use to defend itself from this attack.

Syn cookies

5. [2 points] Can FreeMail use a stateful **packet-filter firewall** to defend itself against the SYN flooding-based DDoS? If so, describe what sort of rule or rules the firewall would need to apply, and what "collateral damage" the rules would incur. If not, explain why not.

Possible solutions:
(1) If the flood uses a fixed number (not too large) of IP source addresses in its packets, then the target could install a number of firewall rules that deny traffic from those addresses. In this case, the collateral damage depends on how much legitimate traffic also comes from those addresses.
(2) If the flood uses a very large number of IP source addresses, either by employing a large number of different systems ("bots") to send the traffic, or by spoofing the IP source address in each SYN packet, it is not feasible to defend against the attack. The target cannot use a rule such as "drop any incoming TCP SYN sent to our web server" without enabling the attack to fully succeed, i.e., the collateral damage would be that no legitimate traffic can reach the server.

6. [2 points] Explain how the FreeMail service could itself be used to mount a DoS attack and how a victim can defend itself.

An attacker could send a large number of bogus spam reports to FreeMail, falsely indicating some victim site V has been sending spam. FreeMail's servers will then visit V to lodge complaints, overwhelming V in the process if the volume of visits is high enough. A victim can defend itself by blocking Freemail IP address.