# Computer Security Exam

Professors M. Carminati & S. Zanero

Milan, 17/07/2018

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.

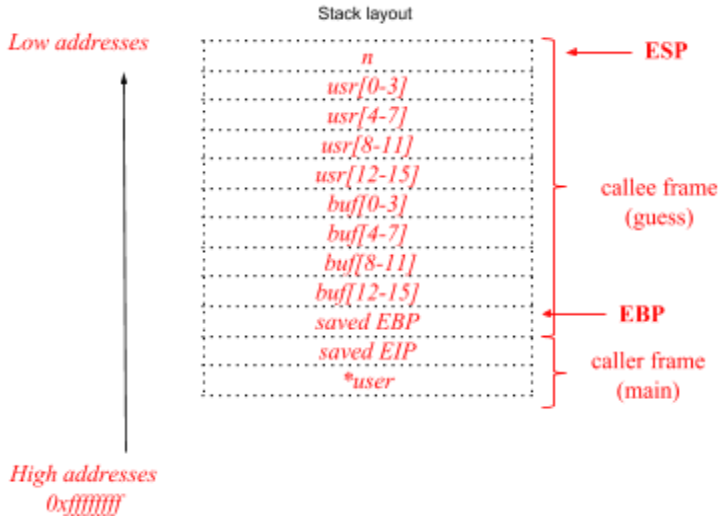
```
00   #include <stdio.h>
01  #include <stdlib.h>
02  #include <string.h>
03
04  int guess(char *user) {
05      struct {
06          int n;
08          char usr[16];
09          char buf[16];
10      } s;
11
12      snprintf(s.usr, 16, "%s", user);
13
14      do{
15          scanf("%s", s.buf);
16          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("Sorry User: ");
26                  printf(s.usr);
27                  printf("\nThe secret is wrong! \n");
28                  abort();
29              }
30          }
31      } while(strncmp(s.buf, "DEBUG", 5) == 0);
32  }
33
34  int main(int argc, char** argv) {
35      guess(argv[1]);
36  }
```

**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 11 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. [1 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.

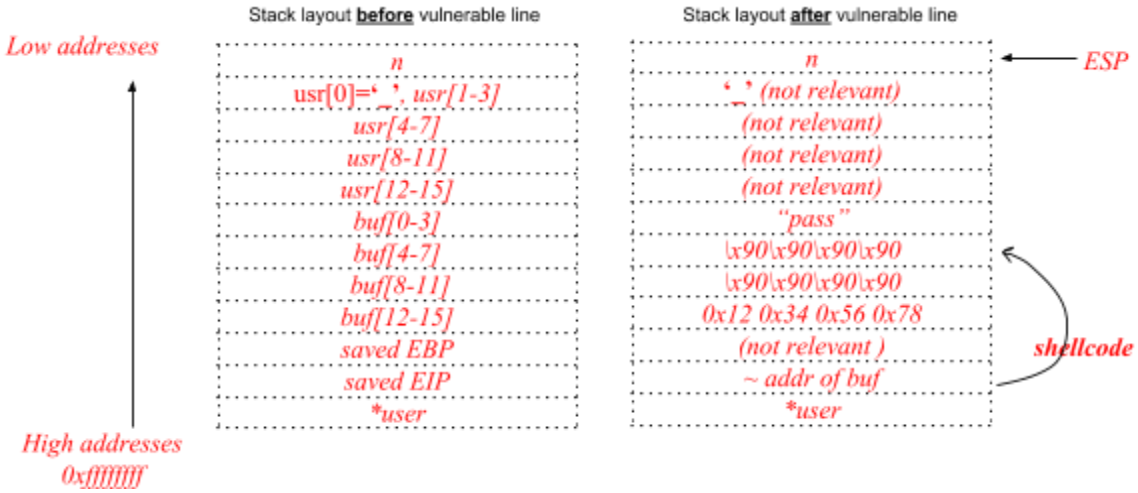
| **Vulnerability** | **Line** | **Motivation** |
|---|---|---|
| Buffer Overflow | *Line 15* | *the scanf reads an user-supplied string of arbitrary length and copies it in a stack buffer* |
| Format String | *Line 26* | *printf(s.usr) where the format string, s.usr, is directly supplied by the (untrusted) user from CLI argument.* |

**3. [3 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 the buffer overflow vulnerability in the above program to execute the following simple shellcode, composed only by 4 instructions (4 bytes): **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).

Command line argument (argv[1]): *any string beginning with '_' is OK*




**4. [2 points]** Now focus on the format string vulnerability you identified. We want to exploit this vulnerability to overwrite the saved EIP with the address of the environment variable $EGG that contain your executable shellcode. Assuming we know that:

- The address of $EGG (i.e., **what to write**) is `0x44674234` (`0x4467(hex) = 17511(dec), 0x4234(hex) = 16948(dec)`)
- The target address of the EIP (i.e., the address **where we want to write** ) is 0x42414515
- The **displacement on the stack** of your format string is equal to 7

write an exploit for the format string vulnerability to execute the shellcode in the environment variable EGG.
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 0x44674234 (<tgt>) to 0x42414515*
*- 0x4467 > 0x4234 -> we write 0x4234 first*
*The value of the command line argument will have the form*
*<tgt><tgt+2>%<N1>c%<pos>$hn%<N2>c%<pos+1>$hn*

*where*
*<tgt> = 0x42414515*
*<tgt+2> = 0x42414517*
*<pos> = 7*
*<N1> = dec(0x4234) - 8 (bytes already written) = 16948 - 8 = 16940*
*<N2> = dec(0x4467) - 16953 (bytes already written) = 17511 - 16948 = 563*
*thus the final string is*

*\x15\x45\x41\x42\x17\x45\x41\x42%16940c%7$hn%563c%8$hn*

**4. [2 points]** Now consider that the program is compiled enabling the exploit mitigation technique known as *stack canary*. Assume that the compiler and the runtime correctly implement this technique with a random value changed at every program execution. Are the two exploits you wrote still working *without modifications*? Why? If not working modify the exploit so that it works reliably in this setting (i.e., enabled stack canaries). Please describe precisely how you would modify the above exploit, and any further assumption you need to make it work. If your exploit needs to perform multiple iteration of any loop, please describe what you would write to the standard input at every iteration of the loop(s).
If the exploit is working without modification, please describe precisely why the vulnerability is still exploitable.

First exploit
*No. the first exploit would overwrite the stack canary (placed in the stack before the return address). The code placed by the compiler in the function epilogue would detect the canary overwrite and abort the execution of the program without jumping to the saved return address, and without executing the attacker's shellcode. We need to make the program perform two iterations of the loop.*
*First iteration: write to the stdin '*`_dbg!\n`*' as argv[1] and 'DEBUG' at line 16; the code at lines 17-21 will print n words from the stack. Assuming that n is sufficiently high to print also the word containing the stack canary (or, if the attacker controls the command line arguments, passing a sufficiently high n), this would leak the stack canary value for that specific execution of the program.*
*Then, as strncmp(s.buf, "DEBUG", 5) == 0 , the* `do...while` *loop will not exit.*
*Second iteration: at this point, we know the value of the stack canary (read during the first loop iteration). We write to stdin (i.e., we put in the buffer) the same exploit I wrote at point 2, changing it slightly so that the stack canary gets overwritten by the value leaked during the first iteration. Also, we need to ensure that strncmp(s.buf, "DEBUG", 5) == 1 so that the loop ends, the function returns, and the shellcode is executed.*
Second exploit
*Yes, the second exploit still work without modification...*

## Question 2 (8 points)

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: handler for the GET request /comments?id=<id>00
A.01    var id = request.get['id'];
A.02    var prep_query = prepared_statement("SELECT username FROM users WHERE id=? LIMIT 1");
A.03    var username = query(prep_query, id);
A.04    var prep_query = prepared_statement("SELECT * FROM comments WHERE username=?");
A.04    var comments = query(prep_query, username);
A.06    for comment in comments {
A.07            echo htmlentities(comment);
A.08    }
Login: handler for the POST request /login
B.01    var password = md5(request.post['password']);
B.02    var username = request.post['username'];
B.03    var prep_query = prepared_statement("SELECT username FROM users
B.04                                        WHERE username=? AND password=? LIMIT 1");
B.05    var result = query(prep_query, username, password);
B.06    if (result) {
B.07            session.set('username', username);
B.08            echo "Logged in.";
B.09    } else {
B.10            echo "User" + username + "does not exists!";
B.11    }
Write comment: handler for the GET request /write?comment=<text_of_the_comment>
C.01    var username = session.get['username'];        // You need to be logged in
C.02    var comment = request.get['comment'];
C.03    var res = query("INSERT INTO comments (username, comment, timestamp)
C.04                                VALUES ( “ +  username + ” , “+  comment + ” , NOW())");
C.05            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., `<`, `>`, `&quot;` and `&apos;` respectively).

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

**users**

| *id* | *name* | *password* |
|---|---|---|
| 1 | Rick | 76ceaaa34826979e77 |
| 2 | Marcello | 563c39089151f9df26 |
| 3 | admin | d1e576b71ccef5978d |

**comments**

| *id* | *user* | *comment* | *timestamp* |
|---|---|---|---|
| 1 | admin | “Welcome” | 03:23:21<br>24/12/2000 |
| 2 | Marcello | “malcontento...” | 18:31:62<br>17/02/2015 |

**1. [4 points]** Only considering code snippet of all the pages**,** identify which of the following web application vulnerabilities are present:

| *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. Specify also the vulnerable lines involved.* | ***If the vulnerability is present in the code above**, explain the simplest procedure to remove this vulnerability. Otherwise write **"No vulnerability"*** |
|---|---|---|
| **Stored cross-site scripting (XSS)** | **Lines:**<br>No, ... | No vulnerability |
| **Reflected cross-site scripting (XSS)** | **Lines: B.10**<br>*Yes, 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.* |
| **Cross site request forgery (CSRF)** | **Lines: C01-C04**<br>*Yes, 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']. 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** | **Lines: C0.3-C0.4**<br>Yes, ... | *The simplest procedure to prevent this vulnerability is to apply escaping/filtering to the "comment/username" variable.* |

**2. [2 points]** 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.

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

**3. [1 point]** 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?

*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).*

**4 [1 point].** Consider the implementation of the session management mechanism:

```
function session.set(key, value) {
    response.add_header("Set-Cookie: " + key + "=" + value);
}
```

Likewise, the `session.get()` function parses the Cookie header in the HTTP request and returns the value of the cookie with the specified key. Describe how an attacker can exploit a vulnerability in this implementation to authenticate as an existing user, and suggest a way to change the function `session.set()` to fix this vulnerability.

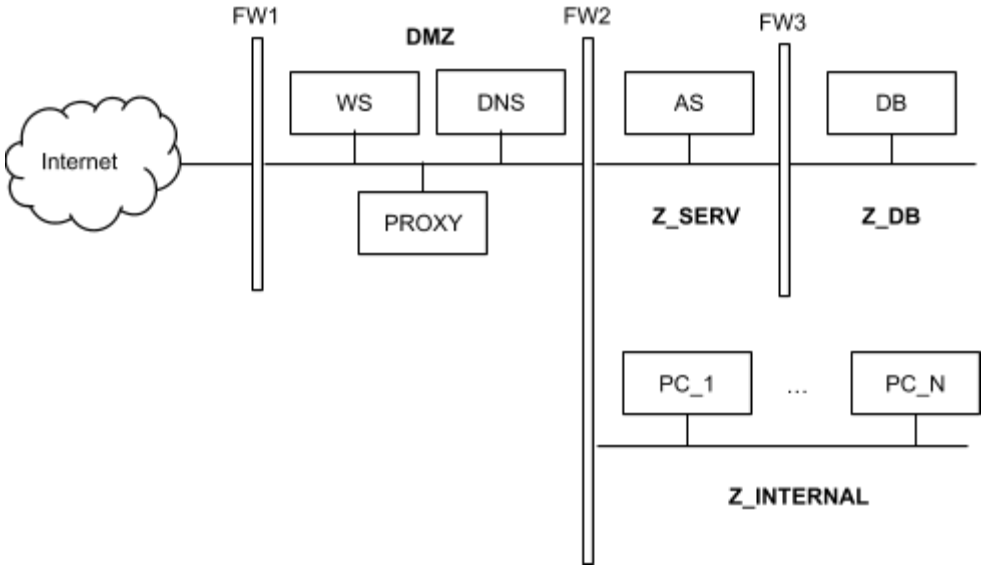
Cookie: username=any existing username

Fix:

- Encrypt the cookie with a key stored on the server (with a nonce and an expiration to avoid replay attacks)
- Store the session data somewhere (e.g., file, database, …) indexed by a random value and set the random value in the cookie instead of the username

## Question 3 (10 points)

Dumb LLC is a software development startup. Their highly secure network contains a web server, that needs to be accessible via HTTP from the Internet and from the corporate network. The web server accesses services exposed by an application server, which, in turn, connects to a local database server. Employees do not need to access neither the application server, nor the database server from their computers (PC_1, …, PC_N). For security reasons, employees can access the Internet only over HTTP and HTTPS, and only through a web proxy that performs filtering and packet inspection. Instead, they can directly access the web server (that also hosts the corporate Intranet) and they need to resolve domain names (including remote ones) through the DNS server.
The network layout is depicted below:




**1. [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, FW2, FW3* | *ANY* | *ANY* | *ANY* | *ANY* | *ANY* | *DENY* | *DEFAULT DENY* |
| *FW1* | *ANY* | *ANY* | *INTERNET--> DMZ* | *WS_IP* | *80* | *ALLOW* | *HTTP* |
| *FW1* | *DNS_IP* | *ANY* | *DMZ-->INTERNET* | *ANY* | *53* | *ALLOW* | *DNS EXTERNAL* |
| *FW1* | *PROXY_IP* | *ANY* | *DMZ-->INTERNET* | *ANY* | *80, 443* | *ALLOW* | *WEB PROXY* |

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| *FW2* | *ANY* | *ANY* | *Z_INTERNAL->DMZ* | *PROXY* | *80, 443* | *ALLOW* | *WEB PROXY* |
| *FW2* | *ANY* | *ANY* | *Z_INTERNAL->DMZ* | *DNS* | *53* | *ALLOW* | *DNS* |
| *FW2* | *ANY* | *ANY* | *Z_INTERNAL->WS* | *WS* | *80* | *ALLOW* | *INTRANET* |
| *FW2* | *WS* | *ANY* | *DMZ-->Z_SERV* | *AS* | *AS_PORT* | *ALLOW* | *WS->AS* |
| *FW3* | *AS* | *ANY* | *Z_SERV-->Z_DB* | *DB* | *DB_PORT* | *ALLOW* | *AS-->DB* |

**2.** Our network analysts captured the following traffic on the computer used by one of our employees (PC_1 in Z_internal), with IP address 192.168.1.6 and MAC address dc:a9:04:7a:ce:29. During the capture, the employee was logging into the Intranet site hosted by the corporate Web server located in the DMZ (WS), with IP address 18.194.76.151 and MAC address dc:a6:03:01:02:fe. Furthermore, we know that the IP address on Z_INTERNAL of the gateway between Z_INTERNAL and the DMZ 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 /login?user=.../pass=...*
18.194.76.151 (38:60:77:b9:79:98) → 192.168.1.6 (dc:a9:04:7a:ce:29) **TCP** *HTTP 200 OK ...*

**2.1 [2 point].** Describe the attack going on in the network, specifying the name and providing a short explanation of how the attack works.

*ARP spoofing \ ARP cache poisoning. Description: see slides.*

**2.2 [1 point].** What is the goal of the attack, in this specific case? Motivate your answer.

*Traffic interception, e.g., to intercept the credentials to the Internet site. It is not denial of service as the traffic continues to flow after the attacker spoofs the gateway.*

**2.3 [1 point].** Can you tell the IP address of the attacker? And the MAC address?

*IP: no*

*MAC: it is 38:60:77:b9:79:98 but it can be easily spoofed*

**2.4 [1 point].** Given the above packet capture, can you tell to what network zone the attacker is connected?

*The attacker is connected to the internal zone .....*

**2.5 [1 point].** Now the Intranet service is migrated to HTTPS, and the employee's browsers correctly validate the server's TLS certificate. Would an attack using this technique lead the same result? Why?

*The attacker is able to intercept the traffic also in this case but, if the client correctly performs certificate validation and the attacker is not able to generate a trusted certificate for the web server, the attacker is not able to intercept the payload of the request.*

**2.6 [1 point].** Assume that you are the network administrator, and control all the switches and routers in the above network, i.e., you can perform filtering and analyses on all packets transiting the networking equipments at the appropriate layers. Suggest a way to (a) detect and (b) prevent this class of attacks.

***Detect**: in general, it is enough to detect duplicate ARP responses with different MAC addresses at the network switches. Alternatively, given the fact that the target is the gateway and that the network admin controls the gateway, detect any ARP response with the IP address of the gateway and raise an alarm.*
***Prevent**: If we only consider ARP spoofing from the gateway, the switches can block any ARP response with the gateway's IP address and from a MAC address different than the gateway's own address. More in general, assuming that the network is configured with DHCP, network switches can listen for DHCP traffic at each port, and store the (IP, mac) tuple in a centralized way (clients can move between switches\ports without renewing the DHCP lease). Then, drop each ARP packet that does not match this tuple. In any case, preventing this attack imposes an increased load on the network switches. This mitigation is actually implemented in some high-end switches (e.g., Cisco's Dynamic ARP Inspection).*

# Question 4 (5 points)

**1. [3 points]** Consider **computer malware**. Please describe the difference between a **virus**, a **worm** and a **trojan**.

| | |
|---|---|
| **Virus** | *See slides* |
| **Worm** | *See slides* |
| **Trojan** | *See slides* |

**2. [2 points]** 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.

We made a binary diff of the two samples in order to evaluate the difference in their layout and reported only the differences here:

**Sample 1**

```
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
```

**Sample 2**

```
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)
```

Additionally, we executed the samples and collected the syscalls in order to evaluate their behaviour (i.e., semantics).

**Sample 1 Trace**

```
execve("./sample1", ["./sample1"], 0x7ffe4fc39080
/* 60 vars */) = 0
brk(NULL = 0x558154aa3000
access("/etc/ld.so.preload", R_OK)      = -1
ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache",
O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=232207,
...}) = 0
mmap(NULL, 232207, PROT_READ, MAP_PRIVATE, 3, 0) =
0x7f0157a64000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/libc.so.6",
O_RDONLY|O_CLOEXEC) = 3
read(3,
"\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20
01\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2105608,
...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7f0157a62000
                        [...]
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7f0157a634c0) = 0
mprotect(0x7f015786f000, 16384, PROT_READ) = 0
                        [...]
munmap(0x7f0157a64000, 232207)          = 0
mmap(NULL, 4, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_SHARED|MAP_ANONYMOUS, -1, 0) = 0x7f0157a9c000
execveat(1852400175, "/bin//sh", NULL, NULL, 0) =0
```

**Sample 2 Trace**

```
execve("./sample2", ["./sample2"], 0x7ffe4fc39080
/* 60 vars */) = 0
brk(NULL) = 0x5575cc15d000
access("/etc/ld.so.preload", R_OK)      = -1
ENOENT (No such file or directory)
openat(AT_FDCWD, "/etc/ld.so.cache",
O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=232207,
...}) = 0
mmap(NULL, 232207, PROT_READ, MAP_PRIVATE, 3, 0) =
0x7fbb4f1f2000
close(3)                                = 0
openat(AT_FDCWD, "/usr/lib/libc.so.6",
O_RDONLY|O_CLOEXEC) = 3
read(3,
"\177ELF\2\1\1\3\0\0\0\0\0\0\0\0\3\0>\0\1\0\0\0\20
01\2\0\0\0\0\0"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=2105608,
...}) = 0
mmap(NULL, 8192, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fbb4f1f0000
                        [...]
close(3)                                = 0
arch_prctl(ARCH_SET_FS, 0x7fbb4f1f14c0) = 0
mprotect(0x7fbb4effd000, 16384, PROT_READ) = 0
                        [...]
munmap(0x7fbb4f1f2000, 232207)          = 0
mmap(NULL, 4, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_SHARED|MAP_ANONYMOUS, -1, 0) = 0x7fbb4f22a000
execveat(1852400175, "/bin//sh", NULL, NULL, 0) =0
```

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

*Polymorphism. See slides.*