Service-Knocking Communication

Santiago Alessandri[1], Matias Fontanini[2], and Nicolás Macia[3]

[1] CERT, CeSPI, Universidad Nacional de La Plata, Argentina
[2] CERT, CeSPI, Universidad Nacional de La Plata, Argentina
[3] Linti, Facultad de Informática, Universidad Nacional de La Plata, Argentina

Abstract. Las herramientas de acceso remoto son un elemento fundamental para la actividad de los administradores de servidores. Debido a esto, es necesario que dichos servidores ejecuten servicios de administración remota, los cuales en general admiten conexiones desde cualquier IP. Esta configuración puede traer problemas de seguridad, como ser ataques de fuerza bruta[1] o ataques a vulnerabilidades específicas de los servicios de administración[2].
En este paper, se presenta una técnica que permite la comunicación remota con aplicaciones corriendo en un servidor sin que las mismas dispongan de un puerto abierto de comunicación. Se camufla la comunicación a través del puerto de un servicio de acceso público en el servidor. Se implementó una herramiena simple como prueba de concepto de la técnica presentada.
Esta técnica permite la comunicación con un servicio sin puertos dificultando la detección del mismo, por lo que llevar a cabo ataques contra el servicio se torna más difícil aún.

Abstract. Administrators need to have a remote access to a server in order to perform administrative tasks. Having an administrative service publicly available arises several security issues, ranging from brute-force attacks[1] to exploiting service's vulnerabilities[2].
In this paper we present a technique along with a simple tool which enables the administrators to communicate with an application with *no open ports* through other running service's open port in order to disguise its traffic.
A direct consequence of the application of this technique is that one could run an almost completely hidden service making it highly difficult to detect, thus to attack.

1 Introduction

Administration services have always been a target chosen by attackers to gain access to a host. Such services should be safe enough to be publicly accessible without compromising the host's integrity. However, this cannot be always ensured.

A public service, just for the fact of being public, reveals an amount of information that can be useful for an attacker. She/he can use this to perform specific

attacks to this service, such as man-in-the-middle, brute-force, exploitation of known vulnerabilities or even using 0-days attacks.

A way of avoiding the aforementioned problem is to hide the service and make it accessible only when it is needed. To achieve this, the *port-knocking*[3] technique has been developed.

Port-Knocking is *"A method for delivery of information via closed ports on a networked computer"*[4]. This technique serves its purpose but it has some limitations.

As this method is based on sending a sequence of packets to different closed ports to trigger an action, it may have the following issues:

- **Host behind a firewall:** Usually the only traffic allowed to pass through the firewall is the one whose destination is the host's public open services. However, the host's administrator does not always have *root* privileges in the firewall, making it impossible to reach the closed ports, thus, not being able to apply *port-knocking*.
- **Host behind a NATed network:** The problem is the same as the one above. Ports are not reachable unless they are forwarded, and even if it was possible, forwarding closed ports is not a reasonable thing to do.

We developed a technique called *service-knocking* which allows communicating with an application with no open ports, avoiding the aforementioned problems. To achieve this, *service-knocking* uses a monitor on other application's open port looking for a sequence of specially crafted packets while *port-knocking* expects a sequence of packets sent to closed ports. Both, *port-knocking* and *service-knocking* trigger an action when the sequence is detected.

As a proof-of-concept we develop and present a simple tool using *service-knocking* to establish a reverse *SSH* connection that allowing the administrator to login on the server without exposing a public administration service.

2 Service-Knocking

Service-Knocking is a technique which involves expecting a sequence of specially crafted packets on the same port where a public service is running, without interfering with its normal behaviour.

In order to implement it, it is necessary to have a public service running on the server. This is going to serve as the inbound channel from which the packets are going to be received.

After having identified the inbound channel, it is needed to set up a packet monitor, analysing the traffic which is directed to it. Traffic analysis consists of searching for a packet whose structure matches the format of the sequence's expected packet.

The structure of the packet is one of the implementation's key points and has to be predefined regarding certain properties. This is going to be developed in detail in the *Packet Structure* subsection.

After a valid sequence of packets is found, a predefined action will be executed. This action may even be determined by the content of the packets received.

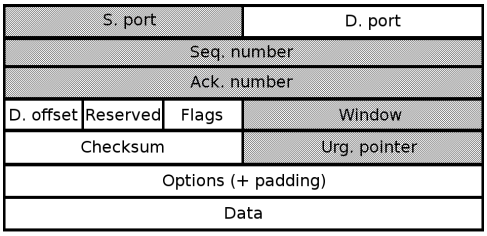

Fig. 1. TCP Segment's customizable fields

2.1 Packet Monitoring

This is the first and one of the most important parts of this technique.

We require a tool or library to capture, monitor, and process the incoming traffic. *libpcap*[5] is an example of this kind of libraries.

There is no need to set up the network device in promiscuous mode because it is not necessary, it will increase the amount of packets to be processed and this can deteriorate the server's performance.

With the purpose of minimizing the amount of packets to analyse, a filter should be applied so that only the packets whose destination port is the chosen service's port are the only ones left.

Since this process is passive, it does not interfere with the normal workflow of the service running on that port.

2.2 Packet Structure

The data that is going to be sent to the hidden service ought to be included inside the transport-layer's PDU's compulsory fields and not in the data field. This should be done to minimize the packet's size, thus reducing the traffic load.

Since the *service-knocking daemon* is going to be monitoring a port where a service is running, there might be high traffic, it is really important not to interpret a normal application packet as one being sent to the monitor service. Achieving this is easier using a TCP segment[6] than using an UDP datagram[6], as the first one contains more *customizable* fields.

In figure 1, the highlighted fields can contain arbitrary values. These fields cannot be analysed for filtering because of their nature:

- **Source Port:** 2 bytes that unless you are expecting a response from the server, can be modified to contain any value.
- **Sequence Number:** 4 bytes that are generated randomly, so there is no way of determining if this value is correct or not.
- **Acknowledge Number:** 4 bytes that have no special meaning while there is no established connection.
- **Window Size:** 2 bytes that, as there is no connection established, can be given any value.
- **Urgent Pointer** 2 bytes which are not used because the *URG* flag is not set.

2.3 Packet Sequence

The sequence of packets that trigger the action should be carefully chosen, since it makes a great impact on the *service-knocking daemon's* reliability.

When choosing this sequence, one must take into account the following properties:

Sequence Length The sequence length is the number of packets required to set off the monitor's action. It depends on the data to be sent and how much of it will be sent in each packet.

On one hand, using a short sequence raises the chances of accepting a series that was not addressed to the *service-knocking daemon*. On the other, although a longer one reduces these chances, it should be kept in mind that since a connection-less communication (the packet is sent without previously establishing the connection) is being used, the risk of packet loss and of disordered arrival of packets increases.

Sequence Order The implementation of the sequence must contemplate the fact that for each packet of the series, one should be able to determine which position it belongs to.

Achieving this is of great importance due to the fact that the communication is unreliable and making a mistake in the order of the packets could trigger an unwanted action with unknown results.

A magic-number mechanism can be used to identify the packet in the sequence.

Packet Uniqueness Identifying that a packet belongs to the sequence and is not part of the service's normal traffic is a key feature to ensure that the *service-knocking daemon's* functionality is reliable.

In order to guarantee the uniqueness of each packet, several measures can be taken:

- **Usage of magic numbers**
- **Inter-field dependencies:** Making a field's value depend on the content of another.
- **Inter-packet dependencies:** The values of a packet field depends on a previous one.
- **Leave the least bits to randomness:** Apply these techniques to the most number of bytes.

Sequence Reset Since unreliable communication is being used and there is no certainty that a normal packet will not be interpreted as part of the sequence, a mechanism to reset it should be implemented. It is not desirable that the *service-knocking daemon* locks itself waiting for a certain packet that will never arrive.

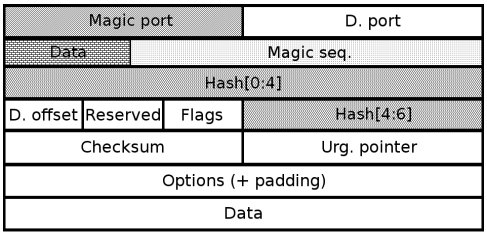

Fig. 2. Daemon's view of TCP segment header

A method that can be easily implemented is setting a time limit for the arrival of packets from the same sequence. If a packet takes longer to arrive than the time limit, the sequence is reset and a new sequence of packets is expected.

3 Proof Of Concept: Reverse SSH

As a proof of concept, we have developed a *service-knocking daemon* which, after a sequence of packets, creates a reverse *SSH* connection on the IP and port given in the sequence. To be able to perform this *proof-of-concept*, an RSA key pair should be generated on the server[7], and the public one should be in the client computer. Using this service, an administrator could perform administrative actions on a host without running any kind of public service to allow this. In addition, it is an extremely safe mechanism, since the RSA key is needed to establish the connection, and afterwards, normal authentication is required.

It is important to notice that our application of *service-knocking* is succeptible to *replay attacks*, but this is not an inherent property of *service-knocking*.

The sources developed to back up this proof of concept are located in `http://www.cert.unlp.edu.ar/uploads/skdaemon/skdaemon.tar.gz`

3.1 Packet Sequence

We have chosen a sequence of 6 packets to trigger the action. In each packet only one byte of data is sent. The first 4 bytes correspond to the IP where the reverse ssh connection will be directed and the last two identify the port to connect to.

A 10 second-threshold between each packet was established. If the threshold is reached, the sequence is reset. This means that the packets that have arrived are discarded and a new sequence is expected.

3.2 Packet Structure

The daemon will be expecting the crafted packets in TCP headers. Figure 2 displays the chosen structure.

- **Magic Port:** In the source port of the TCP header, a 2 byte magic number is expected. This is constant and unique for each of the six packets of the sequence, and is used to identify the packet's position in the series.
- **Data Byte:** The first byte of the *Sequence Number* of the TCP segment contains the actual data being sent to the *service-knocking daemon*.
- **Magic Sequence:** The last 3 bytes of the *Sequence Number* are used in the same way as the magic port. A constant is expected depending on the relative position of the packet in the sequence.
- **Acknowledgement Number:** The *Acknowledgement number* contains the first 4 bytes of the data's MD5 hash. This is done to gain uniqueness and data integrity.
- **Window field:** This 2-byte field contains the 5^{th} and 6^{th} bytes of the data's MD5 hash. It is used for the same purpose as the *Acknowledgement number*.

3.3 Packet Uniqueness

Taking into account the accepted values for each field of the packet, we can calculate the probability of receiving a random packet and interpreting it as a valid one.

The following equations measure the probability for the magic port(1), the magic sequence(2), the 6 bytes of the MD5 hash(3) and the whole packet to be valid using random data(4).

$$mp = \frac{1 \text{ valid value}}{2^{16} \text{ possible values}} = \frac{1}{2^{16}} \tag{1}$$

$$ms = \frac{1 \text{ valid value}}{2^{24} \text{ possible values}} = \frac{1}{2^{24}} \tag{2}$$

$$h = \frac{1 \text{ valid value}}{2^{48} \text{ possible values}} = \frac{1}{2^{48}} \tag{3}$$

$$\text{whole packet} = mp * ms * h = \frac{1}{2^{16}} * \frac{1}{2^{24}} * \frac{1}{2^{48}} = \frac{1}{2^{88}} \tag{4}$$

The probability of a random packet to be taken into account by the daemon is negligible.

3.4 Sequence Uniqueness

To calculate the probability of getting an accepted sequence by generating random packets we have to take into account the following items:

- **Probability of each random packet being valid:** This is the value calculated in the previous section. As each packet has the same structure and is different from each other, then the probability is equal for every packet in the sequence.

– **The number of packets that can arrive in the 10-second time lapse:** Since the daemon has a threshold of 10 seconds; it is necessary to know the number of packets that can arrive in that lapse in order to accurately calculate the probabilities of receiving a valid packet.
The number of packets depends on the network bandwidth. In this case we will analyse it under a gigabit network. The minimal size of a TCP packet is 58 bytes(18 from the Ethernet frame[6], 20 from the IP packet[6] and 20 more from the TCP segment[6]), a gigabit network can transmit $\frac{1024^3}{8} = 134217728$ bytes per second[8]. Therefore, the maximum number of packets that can arrive in 10 seconds is $\frac{134217728}{58} * 10 = 23140980$.

The probability of receiving a valid first packet is $\frac{1}{2^{88}}$ and for each of the following is $\frac{23140980}{2^{88}}$. As a consequence the chances of receiving a valid sequence in a gigabit network is:

$$\frac{1}{2^{88}} * (\frac{23140980}{2^{88}})^5 = \frac{23140980^5}{2^{528}} \approx 7.55215 * 10^{-123} \quad (5)$$

3.5 *Service-knocking daemon*

The application is written in *C* and uses *libpcap*[5] to do the packet monitoring.

When a valid sequence is received, the *Service-Knocking daemon* logs into the ssh service located at <ip_received>:<port_received> authenticating with the server's RSA key.

A reverse ssh tunnel is set up in the client's 8080 tcp port. This is done executing the following command: *ssh -R 8080:localhost:22 <ip_received> -p<port_received>*. In this case it logs in as *root*, but this can be changed by modifying the command used, so that a less-privileged user is used.

Using the previously mentioned tunnel, the administrator is able to reach the server's hidden ssh service to log in with its username and password.

3.6 Client program

The client was written in *Python*[9] using the *Scapy*[10] library to forge the packets.

In order to perform the packet crafting, the client program needs to be run with administrator privileges.

The information needed to send the sequence is taken from 4 arguments:

1. **Destination IP:** The IP where the sequence has to be sent to.
2. **Destination port:** Port number where the daemon is monitoring the traffic.
3. **Reverse SSH destination IP:** IP where the SSH connection has to be sent.
4. **Reverse SSH destination port:** Port where the SSH connection will be established.

4 Conclusion

The presented technique gives administrators a way to communicate with an application with no open ports.

It takes the advantages of *port-knocking* technique while solving its issues like NATs and firewalls preventing reaching the server's ports.

As *Service Knocking* is a flexible technique, it can be adapted to fulfil any requirements needed by the administrator as the action to be performed can be configured.

A direct consequence of the application of this technique is that one could run an almost completely hidden service making it highly difficult to detect, thus to attack.

As pointed out in the *appendix*, it is possible to develop a *Service-Knocking daemon* with a really low false-positive rate ($\approx 7.55215 * 10^{-123}$ in a gigabit network). In addition, even if a random sequence triggers the action this does not expose the server in a significant manner.

References

1. THC-Hydra, network logon cracker *http://thc.org/thc-hydra/*.
2. Richard I Friedber, *OpenSSH Challenge-Response Vulnerability*, SANS GCIH Practical Version 2.1: August 2002.
3. Port knocking, *http://www.portknocking.org*.
4. Ben Maddock, *Port Knocking: An Overview of Concepts, Issues and Implementations*, SANS GIAC GSEC Practical: September, 2004.
5. Pcap Library for C/C++, *http://www.tcpdump.org/*.
6. W. Richard Stevens, *TCP/IP Illustrated: The protocols*, Addison-Wesley Publishing Company, copyright ©1994 Addison Wesley, 0-201-63346-9.
7. Daniel J. Barrett, Richard E. Silverman, Robert G. Byrnes, *SSH, the secure shell: the definitive guide*, O'Reilly Media Inc., copytright ©2005,2001 O'Reilly Media Inc, 0-596-00895-3.
8. Douglas Comer, *Computer networks and internets*, Prentice Hall, 2009, 9780136061274.
9. Python Programming Language, *http://www.python.org*.
10. Scapy Python Library, *http://www.secdev.org/projects/scapy/*.