Examining the generic Socks version 5 proxy protocol

SOCKS FOR PROXY

Socks is a universal proxy protocol for TCP and UDP that allows internal hosts to securely pass the firewall and authenticates users. This article describes the latest version of the Socks proxy protocol and shows how to implement it. **BY THOMAS KUHN AND ACHIM LEITNER**

Many firewall admins allow direct access to the Web from the internal network but are more restrictive with other services such as FTP or SMTP. They rightfully argue that filter rules that allow a minimum of services and ports are easier to track and manage. Application Level Gateways (ALGs) provide even more granular control and are typically implemented as proxies (Figure 1a). However, the application firewall needs a proxy for each service.

The Socks protocol [2] (RFC 1928, Figure 1b) treads a path between the stateful packet filter and the ALG. Socks is implemented in the Dante package [1], for example. The generic Socks proxy technology leaves the firewall in control of applications, separating networks in the Transport Layer and giving clients a fixed request port (typically 1080).

Clients formulate Socks requests, specifying target servers and services (such as HTTP, SMTP, or FTP). The Socks proxy (also known as a Socks server) authenticates the client and authorizes the client for access, sets up the connection to the target server, and transparently forwards any data sent or received.

Intermediate

Normally, client applications need to have integrated Socks support to be able to use the proxy, as Socks does affect the way protocols interact. However, a wrapper can add Socks support to binaries using *LD_PRELOAD* technology. To do this, the wrapper implements a customized socket library.

The name Socks is derived from Socket, the original working title was SOCK-et-S. There are two main versions: Socks v4 and v5. Both protocols insert themselves into the OSI model between the Transport and Application layers. Version 4 is restricted to handling connection requests, honoring Proxy rules, and forwarding application data. It does not provide any kind of authentication and is restricted to TCP. Socks v5 adds robust authentication mechanisms and extends support to UDP.

Roundabout Route

In a typical Socks scenario, the client might want to access the HTTP service provided by a server on an external network. The procedure is shown in Figure 2, the data format in Figure 3, and the field contents are shown in Table 1. The client starts by opening a TCP connection to the Socks proxy (1); the connection uses port 1080 by default. The client sends a Negotiation packet suggesting a few authentication methods (number in *NMETHODS* and methods in *METHODS*).

If the proxy accepts the request (step 2 in Figure 2), it uses a Server Negotiation packet to tell the client its preferred authentication method (*METHODS* with exactly one entry). The proxy then proceeds to authenticate the client (step 3). The exact procedure at this step depends on the selected method.

The client then sends a request to the proxy stating which service it requires (target address *DST.ADDR* and target port *DST.PORT*). The Socks proxy evaluates the request, based on the client ID and the target address, taking an access control list into consideration in a style typical of firewalls. If the client is not allowed the type of access it has requested, the Socks proxy drops the connection to the client.

www.sxc.hu

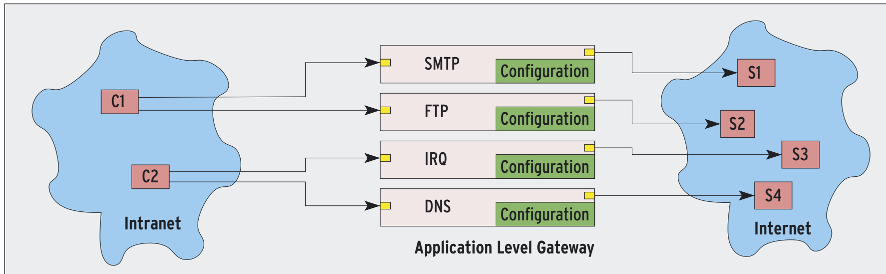

Figure 1a: If the firewall is implemented as an application level gateway, it separates the internal and external networks at application level. However, it then needs a proxy for each protocol.

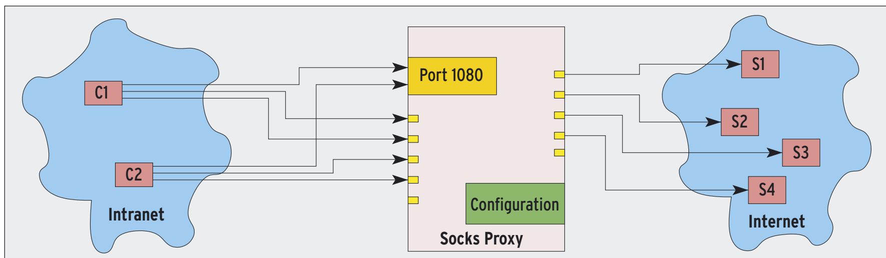

Figure 1b: In contrast to an ALG, Socks assumes the role of a generic proxy, accepting s connections for any application protocols on port 1080, authenticating clients, and authorizing transfers.

In any other case, it replies with one or multiple server reply packets.

Addressed

Socks requests and replies can contain different types of addresses. The protocol supports IPv4 and IPv6 addresses, along with domain names. The latter removes the need for the client to perform a DNS lookup, and the internal network does not need to resolve external DNS names.

Depending on the client request type, that is, depending on the value of *CMD* (Figure 2 and Table 1), the address details in the Socks server reply have a different significance. A reply to a *CONNECT* request contains the *BND.PORT* and *BND.ADDR*, that is the address at which the proxy has connected the target server.

The *BND.ADDR* address is typically not identical to the Socks server address, to which the client sent the original request. This constellation, which is referred to as a multi-homed Socks server, is typical of a Socks firewall that connects two networks. After a successful Connect command, the client and target server can communicate transparently through the proxy; Socks simply forwards any data.

The client sends a *BIND* request to indicate that it expects an incoming connection from a target server. This scenario might seem back-to-front, but it is quite normal in the case of the FTP protocol in active mode. With FTP, and following best client-server traditions, the client first establishes a connection to the FTP server; this is known as the control connection. Whenever a file needs to be transferred, the server establishes a data connection back to the client. Prior to this, the client needs to tell the server which address and port the server should use. Again, this information is sent across the control channel.

Upside-Down World

Socks can selectively allow this type of connection into the internal network. The client opens the control channel to the server by sending a normal Connect request. The client then uses a Bind request within a second connection to ask the Socks proxy to open a port for the incoming data connection.

The proxy sends two replies in response. The first contains the port and address at which the Socks server will listen for the incoming connection. The proxy does not send the second reply until the target server opens a connection. When this happens, the proxy's reply contains the source address and source port the target machine used to open a connection to it. Finally, the proxy forwards the data from the external server to the internal client.

If you want Socks to act as a UDP proxy, the client first needs to use TCP to contact the proxy and authenticate (Figure 4). The *CMD* it stipulates in this case is the third value in Table 1: *UDP Associate*. As the client will actually be using UDP to transmit data later on, it needs to tell the proxy where these packets will be coming from. To do so, the client adds its own address and port to the *DST.ADDR* and *DST.PORT* fields.

The proxy then opens an internal UDP relay port, allowing the client to send packets to the outside world. The client reads the address and port for the relay from the server's reply to the UDP Associate request: *BND.PORT* and *BND.ADDR*. And this is where the client has to send any UDP packets destined for the external network. The client wraps its own UDP packets in a UDP Request (Figure 3 bottom). The UDP Relay stays open for as long as the client keeps the authenticated TCP connection up.

Authentic

The authentication method can also provide trust and integrity between the client and the proxy. The authentication

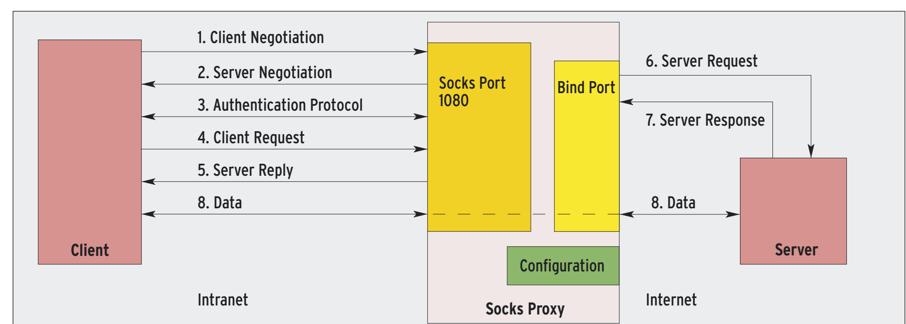

Figure 2: When establishing a Socks v5 connection, the client starts by sending a negotiation packet to the Socks proxy (1). The client authenticates (3); the proxy then establishes the connection to the target server (6) and forwards data (8).

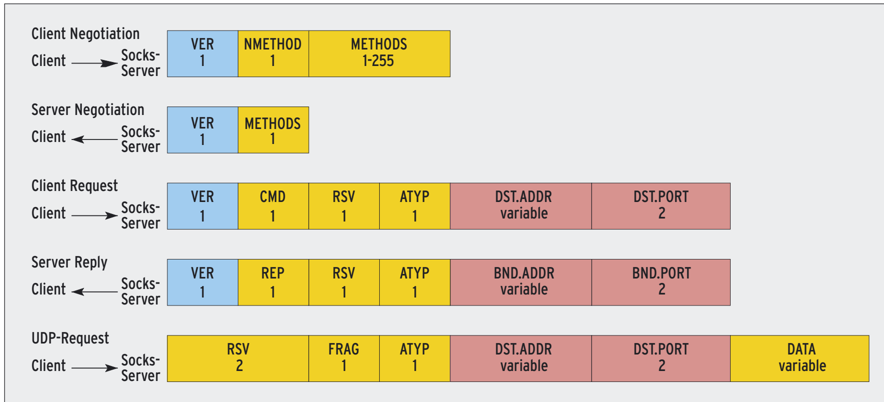

Figure 3: Socks version 5 uses five packet types: Client Negotiation, Server Negotiation, Client Request, Server Reply, and UDP Request. The fields specify the name and size. Table 1 describes the contents.

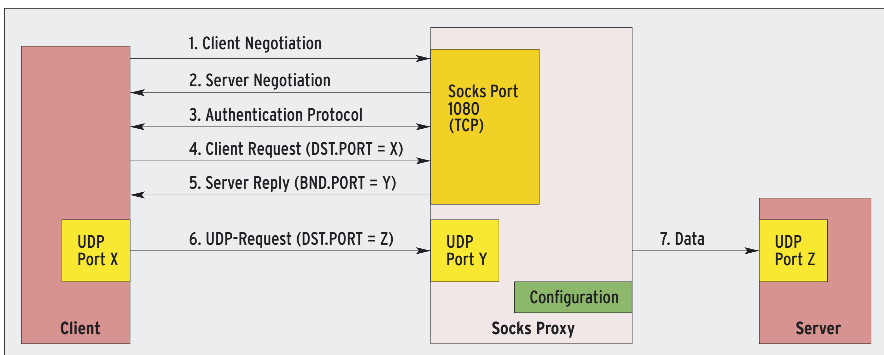

Figure 4: In a UDP scenario, the client first uses TCP to connect to the Socks proxy. The Client Request (4) contains a UDP Associate command, in which the client tells the proxy from where it will be sending UDP packets.

might involve encapsulating the data, for example, using a secure protocol such as SSL or TLS. After completing the client/server negotiation process, the client then authenticates using SSL/TLS. Any other data sent during the connection can also be protected by SSL/TLS, and this form of secure communication ensures trust and integrity. Users can run mobile, wireless devices securely via the Socks proxy.

Dante

The BSD-licensed Socks client and server implementation Dante for Unix [1] supports Socks v4 and v5 and the less common MSproxy. Version 1.1.15 was released at the end of January 2005.

Dante is developed by a Norwegian consultancy called Inferno Nettverk A/S, who also have commercial modules for bandwidth control and port/forwarding monitoring.

This said, the free version is typically fine for most tasks. Besides providing Socks and MSproxy services, it can also act as a HTTP proxy, authenticate users based on usernames and passwords, or via Pluggable Authentication Module. Support for interface names in the configuration file allows it to support DHCP.

Configuring the Proxy

The normal install, using *configure && make && make install*, drops the Socks server configuration file into */etc/sockd.conf* (Listing 1). In line 1, a *logoutput* instruction tells Dante where to send the logfiles (Syslog or Stdout). The internal and external network interfaces are specified by interface names in lines 4 and 5. This is useful for computers with a DHCP-based configuration. Lines 6 and 7 show that IP addresses are just as acceptable. Note that the internal interface needs a port number.

The authentication methods supported by Dante include username / password (line 9), the Ident method as specified in RFC 931 (line 11) and PAM. The Socks server needs different user privileges to reflect the authentication method. If it needs access to the password file, it will opt for a privileged user account (defined as *proxy* in line 14), although it is quite happy to be a *nobody* (line 15) under other circumstances. Best practice would suggest using a dedicated user account for the Socks v5 server. Admins

Listing 1: Socks Server

```
01 logoutput: syslog
02 #logoutput: stdout
03
04 internal: eth0 port = 1080
05 external: eth1
06 #internal: 10.0.0.11 port = 1080
07 #external: 192.168.23.1
08
09 method: username
10 #method: none
11 #method: rfc931
12 #method: pam
13
14 user.privileged: proxy
15 user.notprivileged: nobody
16
17 client pass {
18   from: 10.0.0.3/
   0 port 1-65535 to: 0.0.0.0/0
19 }
20
21 client block {
22   from: 0.0.0.0/0 to: 0.0.0.0/0
23   log: connect error
24 }
25
26 block {
27   from: 0.0.0.0/0 to: 10.0.0.11/0
28   log: connect error
29 }
30
31 pass {
32   from: 10.0.0.3/
   0 to: 10.0.0.10/0
33   protocol: tcp udp
34 }
35
36 block {
37   from: 0.0.0.0/0 to: 0.0.0.0/0
38   log: connect error
39 }
```

can run the server in a chroot jail to keep it well away from system files and also give the server its own password file.

Well Filtered

Filter rules in the configuration files allow you to specify which clients can access the Socks proxy and to which addresses the proxy is allowed to connect. Dante parses the filter rules sequentially. It first evaluates any rules with the *client* prefix to establish which computers are allowed to access the Socks server (Lines 17 through 24). The *pass* rules allow access, whereas *block* rules disallow access. Lines 17 through 19 allow the computer with the IP address of 10.0.0.3 unrestricted access, whereas lines 21 through 24 deny any other access. These rules are applied at TCP/IP level and have nothing to do with the Socks protocol.

The second class of filter rules checks the content of the client requests. These rules specify the kinds of requests the proxy will honor. The *block* rule in lines 26 through 29 of Listing 1 rejects any requests from computers wanting to connect to 10.0.0.11. The TCP and UDP traffic from the host 10.0.0.3 with requests for 10.0.0.10 is permitted (lines 1 through 34). The proxy will ignore any other requests.

Table 1: Packet Tags

Tag	Content/Description
ATYP	Address Type:
	0x01: IPv4 address
	0x02: Domain name
	0x03: IPv6 address
BND.ADDR	Socks Proxy source address for data transfer to server
BND.PORT	*Socks* Proxy source port for data transfer to server
CMD	Transmission types:
	0x01 CONNECT
	0x02 BIND
	0x03 UDP Associate
DST.ADDR	Target address requested (on server)
DST.PORT	Target port requested (on server)
FRAG	Current fragment number (for UDP packets)
METHODS	Selection field for authentication method:
	0x00: No authentication
	0x01: GSSAPI
	0x02: User name and password
	0x03 through 0x7E: Defined by IANA
	0x80 through 0xFE: Reserved for private methods (only used locally)
	0xFF: The proxy has refused the methods offered by the client
NMETHODS	Number of entries in *METHODS field*
REP	Reply field:
	0x00: Successful
	0x01: Generic Socks proxy error
	0x02: Connection disallowed by ruleset
	0x03: Network not accessible
	0x04: Host not accessible
	0x05: Connection request refused
	0x06: Timeout (TTL expired)
	0x07: Socks command not supported
	0x08: Address type not supported
	0x09 through 0xFF: Not defined
RSV	Reserved
VER	Protocol version (0x05)

First Tests

Calling */sbin/sockd -d* launches the proxy in debug mode. Launching in debug mode tells the proxy to log anything important in *logoutput*. Ethereal is perfect for checking the details of the communication. We used the Mozilla browser as our test client. We set the Socks server to 10.0.0.11 and port 1080 in *Manual proxy configuration* in our case.

If the Socks proxy refuses a connection on account of missing or inappropriate access privileges, the user might become aware of the symptoms without ever learning the reason for the connection failure. For example, if the Mozilla browser is faced with a connection failure, it might simply state that *The document contains no data* in case of an error in part 2 of the filter rules, but there is no mention of the proxy being the cause. To investigate the possible causes for a connection problem of this kind, check out the proxy logfiles, which will be in*/var/log/messages* if you use Syslog.

INFO

[1] Dante: *http://www.inet.no/dante/*

[2] RFC 1928, "SOCKS Protocol Version 5": *http://www.ietf.org/rfc/rfc1928.txt*

[3] RFC 1929, "Username/Password Authentication for SOCKS V5": *http://www.ietf.org/rfc/rfc1929.txt*

[4] RFC 1961, "GSS-API Authentication Method for SOCKS Version 5": *http://www.ietf.org/rfc/rfc1961.txt*

Socks All Round

Besides the Socks server, the Dante package has a simple wrapper script called *socksify*. The *socksify* script provides the user with the option of adding Socks capabilities to most network client programs. With *socksify,* you can add Socks capability to protocols such as SMTP, FTP, NTP, DNS, or IRQ. For example:

```
./socksify -c ftp 10.0.0.10
```

In cooperation with a suitable */etc/socks.conf* configuration file, the preceding command tells *socksify* to talk the *ftp* client program into using the Socks proxy without needing to rebuild the client.

```
route {
  from: 0.0.0.0/0 to: ⮐
  0.0.0.0/0 ⮐
  via: 10.0.0.11 port = 1080
  proxyprotocol: socks_v5
}
```

The preceding settings tell *socksify* to use Socks v5 as its proxy protocol, and to establish a secure network connection via port 1080 on the computer at 10.0.0.11.

One for All

The Socks technology gives network admins the ability to deploy a simple and transparent method for security management. Socks also adds authentication and encryption to networked applications. In contrast to many other protocols, the Socks proxy protocol does not separate connection and user authentication, and thus, Socks gives the firewall complete control over all data traffic. ■