Title: Automatic Protocol Identification on Scanned Ports

Author:

Izar Tarandach

Contact Address:

Bindview/Netect Ltd.
55 New York Avenue
Framingham, MA 01701 USA
izar@bindview.com

Abstract:

We examine the problem and propose a solution for automatically identifying the protocol run on ports that a previous network scan found to be open.

0.1 Introduction

A security scanner is supposed to check each and every daemon for known security problems. This assumption carries a price of "we can only check what we see where we expect to see it"; meaning that we can only be guided by a previously-known set of data to establish what server is running on what port.

That work is made somewhat easy by the existance of mechanisms like the portmapper and inetd, byt the adherence to rules like /etc/services and by the use of well-known ports.

But the people that adhere to these rules normally wear white-hats, and from those we can expect cooperation and interest in running the scanner. The job becomes somewhat more complicated when the scanner has to hit a system where crackers have been at work. The disinformation begins by running potentially data-leaking services on unorthodox locations.

In the past years there has been a steep rise in the number of information-related security events that were initiated either from the inside of a protected network or by someone with sporadic or once-upon-a-time access to the network. These may be determined invaders, contractors, trojan-horses/backdoors plants, or even (and in many cases) left-off employees leaving behind a way to reach the network.

So, how do we point a scanner in the direction of services that the administrator decided to put on non-standard ports, so that these services can be checked for the holes known about them, and how can we help an administrator by identifying all those services that may act as stepping stones for those intruders we mentioned ?

Tools like nmap[1] help in mapping open ports in a given host, but to actually identify the protocol being used in the port an administrator would have to go and either directly access that port via the network, more or less "guessing" the protocol by interacting with it, or login into the machine and use a tool like lsof[2] to identify the process holding the port open, or examine the inetd.conf (assuming a Unix box) or the Services dialog (assuming a NT box), or worse, get a list of processes and start killing one by one until the port closes (the massacre solution). There is not one tool that aims to say "Port 12345/udp is running mountd" or "Port 21001/tcp is running ftp".

0.2 The network security aspect

From a security point of view, we are naturally more interested in those protocols that can be used to transfer proprietary information or give access to a system by an unauthorized party. By observing only those, we come to two jumping-points:

- the casual security-breacher will use common protocols like Telnet, FTP or the R-family to grant himself access, normally running on non-priviledged ports (n>1024).
- an advanced intruder may use protocols of his own, inclusive making use of (readily available) encryption tools.

Statistics show [3] that most attackers may be considered to be in the casual-to-moderate skill level. Therefore, the brunt of our solution will try to cover the set of the publicized protocols (based on RFCs[4]). We will try to provide solutions both for the ASCII,dialog-based protocols and for binary ones. Also, we will suggest basic mechanisms to cover the case of unidentified protocols: in the case of unidentified or possibly proprietary protocols, we suggest planting surveilance tools and following easy procedures that will be able to determine the volume and direction of the information flow, with the intention of identifying the intruder (at the possible cost of information).

[1] A tool by Fyodor, used to list open ports and remotely identify the running OS on a network machine. Available at http://www.insecure.org/nmap

[2] Victor A. Abell <abe@purdue.edu> is the author of this great piece of software, that can show information about files opened by processes, where a file can be anything giving a file descriptor in the Unix sense (a regular file, a directory, a block special file, a character special file, an executing text reference, a library, a stream or a Internet socket, NFS file or UNIX domain socket).

[3] (insert some statistics here)

[4] Request For Comments - documents specifying, among other things, the full requirements and behavior of specific protocols

0.3 Methodology

Probably the easiest way to approach the problem is to mimic the administrator's first option and try to guess the protocol by interacting with it.

The first question that pops to mind is, what protocol do we try first ?

One idea is to use very basic simple rules - use the port number to approximate what protocol this may be (reduce the port number to something like what might be found on /etc/services, by removing trailing zeroes (2100→21), removing leading thousands (1021→21), playing with double-numbering (8080→80). The rationale behind this is that for some reason, people seem to like to use this sort of numbering when plugging a daemon on a port that not that one that it was supposed to be. No hard rules here, just observations picked up along the way. So, we hope to reach a valid candidate protocol by deriving the "correct" port number from the real port number, and following the guess to /etc/services, which should give us a protocol name.

If we are not able to reach a decision by applying these simple rules, educated guessing is the next choice. We throw the first, simpler and broader protocols at the port first, Telnet and Rsh.

If these two are not successful, use trial & error - if the protocol is not identified (handshake dialog not successful) move with the error message towards a better guess. This approach seems to be valid either to binary or textual protocols, over TCP and UDP.

Once all options are ran thru and no final result is finished, move to the sniffing step, trying to collect data and alerting a human operator.

We cannot forget that the tool has to deal with protocols that close-on-bad-input, so that a lot of connections may be needed to nail down a given protocol. This tool will not be a speed demon like nmap.

One possible optimization is to capture the initial message of those binary protocols that interrogate (initiate the dialog) and use "templates" to match it. The reason for restricting this to binary protocols is that textual ones can have their banners changed sufficiently at compilation time so that it would be unconclusive to use it. Of course, in textual protocols this can be used as a fallback method to reach a first candidate for testing, since in that case any first try should be as good as any. The templates would use RFC-defined fixed values present on each protocol in order to identify it.

By the way of proof, we can adopt several methods: we can either begin at the assumption that a specific service is running on a port, and move to disprove that it is there (and failing), we can send illegal input (causing it to respond with probably more protocol-specific behaviour), we can try to complete partial handshakes that definitive information about the nature of the protocol, or we can analyze patterns present in the messages we receive from the protocol (mostly for binary ones, so that we don't encounter again the banner confiability problem).

0.4 Implementation

We chose Perl to implement the tool because of its clearity, speed of development and the ready number of protocol implementations available, if necessary, to avoid recoding of complicated negotiation options.

```
$protocol("rsh") = {
     "name" => "rsh",
     "port" => 514,
     "transport" => "tcp",
     "kind" => "binary",
     "validator" => &rsh_validator(),
     "similar" => "rlogin"
};
sub rsh_validator {
    #from port 512<=p1<=1023, send 512<=p2<=1023
    #!= p1,\0. If a connection is made from the target     #to us on port p2, rsh ha
    #if one of them not confirmed, return
    #NOT_THIS_ONE, jump to similar.
}
$protocol("ftp") = {
      "name" => "ftp",
      "port" => 21,
      "transport" => "tcp",
      "kind" = "text",
      "validator" => &tcp_validator(),
      "similar" => "pop"
};
sub ftp_validator {
     # connect to port. record banner but ignore it.
     # Send ‘‘USER xxxxx\n’’
     # expect ‘‘^331*’’, or a ‘‘^230*’’. these are
     # good FTP signs. - if we get a ‘‘+ OK’’ go to        #$similar
};
```

This way, after the port number leads us (or not) to a candidate (say, FTP), we can invoke the $protocol($candidate) object and use its VALIDATOR member to prove for that specific protocol. In case the function cannot prove it, it will call the $protocol($similar) validator, and so on, until all possibilities are exhausted or a match is found.

This gives us a time optimization, since only protocols that have a chance to match are tried. It can be further improved by a quick connect/send garbage/grab response/close cycle, that will give initial information about the kind of protocol, if either binary or text, and provide more information for an educated guess:

```
use port number to aproximate a guess at /etc/services
while (not tried every possible protocol) {
        $v = $protocol{$guess}→validator;
        if ($v == TRUE) {
             return $protocol→name;
        }
        close connection; # return to sane state
        $guess = $protocol{$guess}→$similar;
}
```

Of course we put mechanisms in place to avoid cycling thru protocols already tried for a given port; in case all similarities are tested and found not appropriate, we either

pass on to less probable protocols, or we declare ourselves incapable of determining the protocol and offer to put a sniffing program into the problem.

0.5 If we can't identify it: the sniffing step

In those cases were we can not identify the protocol, a number of measures is suggested: first of all, we put in place a sniffer (like tcpdump or snoop) that will log all data, incoming and outgoing, as well as the connection points (IP numbers and ports) and datestamp them, with the objective of amassing enough data to give an investigator the possibility of figuring out the responsible for the dialog.

In this case, the nature of the transmitted information becomes part of the scenario. So, it would be appropriate to identify the internal focus of the rogue daemon, and attach to it surveilance tools that could give the investigator the actual files being accessed by the unknown daemon running the unidentified protocol. In case the information is too sensitive, the connection may be terminated. If, instead, the loss of information can be tolerated, the knowledge of the actual files being accessed may facilitate the identification of the interested party.

In the example code section we present a simple script that is fed the (local) port where the daemon is running, and using lsof will log the files accessed in real-time.

0.6 Case Study

0.7 Example code