Improving TCP's Resistance to Blind Attacks through Ephemeral Port Randomization

Fernando Gont
Facultad Regional Haedo, Universidad Tecnológica Nacional
Haedo, Provincia de Buenos Aires, Argentina
fgont@frh.utn.edu.ar

Abstract

Recently, awareness has been raised about a number of "blind" attacks that can be performed against the Transmission Control Protocol (TCP) and similar protocols. The consequences of these attacks range from throughput-reduction to broken connections or data corruption. These attacks rely on the attacker's ability to guess or know the four-tuple (Source Address, Destination Address, Source port, Destination Port) that identifies the transport protocol instance to be attacked. While there have been a number of proposals to mitigate these Vulnerabilities, the most obvious mitigation -- TCP port randomization -- has been the one least engineered. In this paper we analyze a number of approaches for the random selection of client port numbers, such that the possibility of an attacker guessing the exact value is reduced. We discuss the potential interoperability problems that may arise from some port randomization algorithms that have been implemented in a number of popular operating systems, and propose a novel port randomization algorithm that provides the obfuscation while avoiding the interoperability problems that may be caused by other approaches. While port randomization is not a replacement for cryptographic methods, the described port number randomization algorithms provide improved security/obfuscation with very little effort and without any key management overhead.

Keywords: Transport protocols, port randomization, obfuscation, blind attacks

1 INTRODUCTION

Recently, awareness has been raised about a number of "blind" attacks that can be performed against the Transmission Control Protocol (TCP) [1] and similar protocols. The consequences of these attacks range from throughput-reduction to broken connections or data corruption [15] [12] [10]. All these attacks rely on the attacker's ability to guess or know the four-tuple (Source Address, Source port, Destination Address, Destination Port) that identifies the transport protocol instance to be attacked.

Generally, the four-tuple required to perform these attacks is not known. However, as discussed in [10] and [12], there are a number of scenarios (notably that of TCP connections established between two BGP routers [13]), in which an attacker may be able to know or guess the four-tuple that identifies a TCP connection. In such a case, if we assume the attacker knows the two systems involved in the TCP connection to be attacked, both the client-side and the server-side IP addresses could be known or be within a reasonable number of possibilities. Furthermore, as most Internet services use the so-called “well-known” ports, only the client port number might need to be guessed. Unfortunately, most systems choose the port numbers they use for outgoing connections (the so-called “ephemeral ports”) from a subset of the whole port number space, and implement ephemeral port selection algorithms that make it trivial for an attacker to guess the port numbers used by clients for outgoing connections.

In this paper we describe a method for random selection of ephemeral ports, thereby reducing the possibility of an off-path attacker guessing the exact value. This is not a replacement for cryptographic methods such as IPsec [7] or the TCP MD5 signature option [6]. However, the proposed algorithm provides improved obfuscation with very little effort and without any key management overhead.

The mechanism described is a local modification that may be incrementally deployed, and does not violate the specifications of any of the transport protocols that may benefit from it [1] [2] [8] [9].

Since the mechanism is an obfuscation technique, focus has been on a reasonable compromise between level of obfuscation and ease of implementation. Thus the algorithm must be computationally efficient, and not require substantial data structures.

2 EPHEMERAL PORTS

2.1 Traditional Ephemeral Port Number Range

The Internet Assigned Numbers Authority (IANA) assigns the unique parameters and values used in protocols developed by the Internet Engineering Task Force (IETF), including well-known ports [11]. The Internet Assigned Number Authority (IANA) has traditionally reserved the following use of the 16-bit port range of TCP and UDP:

- The Well Known Ports, 0 through 1023
- The Registered Ports, 1024 through 49151
- The Dynamic and/or Private Ports, 49152 through 65535

The range for assigned ports managed by the IANA is 0-1023, with the remainder being registered by IANA but not assigned. The ephemeral port range has traditionally consisted of the 49152-65535 range.

2.2 Traditional Ephemeral Port Selection Algorithm

As each communication instance is identified by its four-tuple {local IP address, local port, remote IP address, remote port}, selection ephemeral port numbers must result in a unique four-tuple.

TCP implementations have traditionally implemented a very simple ephemeral port selection algorithm, which simply selects ephemeral ports incrementally. Figure 1 shows the traditional ephemeral port selection algorithm in pseudocode. We will refer to this as 'Algorithm 1'.

```
next_ephemeral = 1024;  /* init., could be random */
count = max_ephemeral - min_ephemeral + 1;

do {
     port = next_ephemeral;

     if (four-tuple is unique)
         return next_ephemeral;

     if (next_ephemeral == max_ephemeral) {
         next_ephemeral = min_ephemeral;
     } else {
         next_ephemeral_port++;
     }

     count--;

} while (count>0);
```

Figure 1: Traditional ephemeral port selection algorithm

A global variable “next_ephemeral” stores the port number that should be selected the next time the port selection function is called. Table 1 shows how the algorithm could possibly select port numbers when a host establishes a number of consecutive connections to both the same and different remote sockets. From the table we note that port numbers are selected incrementally, regardless of the remote TCP socket.

Nr.	IP:port	min_ephemeral	max_ephemeral	next_ephemeral	port
#1	128.0.0.1:80	1024	65535	1024	1024
#2	128.0.0.1:80	1024	65535	1025	1025
#3	170.210.0.1:80	1024	65535	1026	1026
#4	170.210.0.1:80	1024	65535	1027	1027
#5	128.0.0.1:80	1024	65535	1028	1028

Table 1: Sample scenario for the traditional ephemeral port selection algorithm

The algorithm is simple and efficient, and is used in most TCP implementations. However, it has two weaknesses. Firstly, given that it selects port numbers incrementally and independently of the remote TCP socket, the algorithm may quickly cycle through all the port numbers in the ephemeral

port range. This may potentially lead to a port number "collision", that is, the algorithm may select a port number that results in a connection-id that is still in use in the network. Secondly, and most important for the purpose of this article, it reveals information about which port numbers will be selected for future outgoing TCP connections. If an attacker gets to know the port number selected for a recent outgoing TCP connection, he can easily guess the port numbers that will be selected for future outgoing connections.

2.3 Port Number Collisions

While it is possible for the ephemeral port selection algorithm to verify that the selected port number results in connection-id that is not currently in use at that system, there resulting connection-id may still be in use at a remote system. For example, consider a scenario in which a client establishes a TCP connection with a remote web server, and the web server performs the active close on the connection. While the state information for this connection will disappear at the client side (that is, the connection will be moved to the fictional CLOSED state), the connection-id will remain in the TIME-WAIT state at the web server for 2*MSL (Maximum Segment Lifetime). If the same client tried to create a new incarnation of the previous connection (that is, a connection with the same connection-id as the one in the TIME_WAIT state at the server), a port number "collision" would occur. The effect of these port number collisions range from connection-establishment failures to TIME-WAIT state assassination (with the potential of data corruption). In scenarios in which a specific client establishes TCP connections with a specific service at a server, these problems become evident. Therefore, an ephemeral port selection algorithm should ideally lead to a low port reuse frequency, to reduce the chances of port number collisions.

3 PORT RANDOMIZATION

As discussed in Section 1, a simple mitigation approach for all those vulnerabilities that require the attacker to guess or know the four-tuple that identifies the target connection is to obfuscate that four-tuple through a careful selection of the client port number.

There are a number of characteristics that an ideal port obfuscation algorithm should have. Firstly, it should minimize the predictability of the selected port numbers. Ideally, client port numbers should be selected randomly, and thus it would be impossible for an attacker to make an educated guess about the client port number in use by the target TCP connection.

Secondly, it should minimize the port re-use frequency, to avoid interoperability problems. A high port reuse frequency might lead to port number collisions, in which a port number is reused leading to a connection-id that is still in use in the network. These port number collisions lead to interoperability problems (the connection request will fail) which are clearly undesirable.

Finally, the port selection algorithm should avoid selecting port numbers that are needed by popular applications (such as port 80, port 6667, etc.). If a client binds a port number, and that port number is later needed by some application (while the port number is still in use), the application will fail.

3.1 Ephemeral Port Number Range

As mentioned in Section 2.1, the ephemeral port range has traditionally consisted of the 49152-65535 range. However, it should also include the range 1024-49151 range.

Since this range includes user-specific server ports, this may not always be possible, though. A possible workaround for this potential problem would be to maintain in memory an arrays of bits, in which each bit corresponds to each of the ports in the range 1024-65535. A bit set to 0 would indicate that the corresponding port is available for allocation, while a bit set to one would indicate that the port is reserved and cannot be allocated. Thus, before allocating a port, the ephemeral port selection function would check this array of bits, avoiding the allocation of ports that may be needed for specific applications.

Transport protocols should use the largest possible port range, since this improves the obfuscation provided by randomizing the ephemeral ports.

3.2 Ephemeral Port Randomization Algorithms

3.2.1 A Simple Port Randomization Algorithm

In order to address the security issues discussed in Section 2.2, a number of systems have implemented simple port number randomization algorithm, shown in Figure 2. We will refer to this algorithm as 'Algorithm 2'.

```
next_ephemeral = min_ephemeral + random()
                     % (max_ephemeral - min_ephemeral + 1)
count = max_ephemeral - min_ephemeral + 1;

do {
    if(four-tuple is unique)
            return next_ephemeral;

    if (next_ephemeral == max_ephemeral) {
        next_ephemeral = min_ephemeral;
    } else {
        next_ephemeral_port++;
    }

    count--;
} while (count > 0);

return ERROR;
```

Figure 2: Simple port randomization algorithm

This algorithm randomly selects a port number from the range {min_ephemeral, max_ephemeral} and, if the selected port number is in use, tries the next available port number (in the specified port range).

This algorithm is excellent from the point of view of obfuscation, as it selects the client port numbers randomly, making it hard for an attacker to make an educated guess about the client port number in use for the target TCP connection. However, it has a number of weaknesses.

Since this algorithm performs a completely random port selection (i.e., without taking into account the port numbers previously chosen), it has the potential of reusing port numbers too quickly. Consequently multiple ports may have to be tried and verified against all existing connections before a port can be chosen. Although carefully chosen random sources and optimized four-tuple lookup mechanisms (e.g., optimized through hashing), will mitigate the cost of this verification, some systems may still not want to incur this unknown search time.

Additionally, potentially high port reuse frequency might lead to port number collisions at the server side, which would lead to the interoperability problems discussed in Section 3.2 of this paper. Systems that may be specially susceptible to this kind of repeated four-tuple collisions are those that create many connections from a single local IP address to a single service (i.e. both IP addresses and server port are fixed). Gateways such as proxy servers are an example of such a system.

3.2.2 A novel port obfuscation algorithm

Figure 3 shows the pseudocode for a novel port obfuscation algorithm [14], modeled after the ISN (Initial Sequence Number) selection algorithm described in RFC 1948 (“Defending Against Sequence Numbers Attacks”). The algorithm aims to achieve the obfuscation quality of the simple port randomization algorithm described in the previous section, while keeping the port reuse frequency properties of the traditional TCP port selection algorithm.

```
next_ephemeral = 1024;  /*init., could be random */

offset = F(local_IP, remote_IP, remote_port, secret_key);

do {
    port = min_ephemeral + (next_ephemeral + offset)
               % (max_ephemeral - min_ephemeral + 1);
    next_ephemeral++;

    if(four-tuple is unique)
        return port;

    count--;

} while(count > 0);

return ERROR;
```

Figure 3: A Novel Port Obfuscation Algorithm

The strategy to achieve both goals is to separate the port number space for each remote TCP socket, producing a monotonically-increasing port number sequence (with a random initial port number) for each of them. That is, two consecutive connection requests sent to different TCP sockets would use unrelated client port numbers, while two consecutive connection requests to the same TCP endpoint would use incremental port numbers.

Ephemeral port numbers are selected as the sum of the result of a function F() and the variable “next_ephemeral”. F() is a hash function fed with the server TCP socket {server IP address, server TCP port} and a secret key specified by the system administrator or randomly chosen at system startup. This function F() provides a random “offset” that will be different for each remote TCP

socket. On the other hand, “next_ephemeral” is incremented each time the algorithm selects an ephemeral port, thus ensuring that two consecutive outgoing connections will use different ephemeral port numbers.

Thus, F() provides for the obfuscation in the port number selection, while “next_ephemeral” provides for a monotonically-increasing sequence. Provided F() is a cryptographically-secure hash function, and that the attacker does not know the secret key used as input to F(), it will be very difficult for an off-path attacker to guess the ephemeral port number selected for the connection.

Ideally, the algorithm would use one “next_ephemeral” variable for each of the possible results of F(). However, as this would require a probably unacceptable amount of memory, the algorithm uses a single global “next_ephemeral” variable. The downside of this engineering decision is that the selection of a port number in any port number sequence will cause all the other port number sequences to “skip” a port number they could have potentially used, unnecessarily.

An analysis of a sample scenario can help to understand how this algorithm works. For example, let’s suppose that some host tries to establish TCP connections with a number of remote TCP sockets. Table 2 illustrates, for a number of consecutive connection requests, some possible values for each of the variables used in this novel port obfuscation selection algorithm. Additionally, the table shows the result of the port selection function.

Nr.	IP:port	offset	min_ephemeral	max_ephemeral	next_ephemeral	port
#1	128.0.0.1:80	1000	1024	65535	1024	3048
#2	128.0.0.1:80	1000	1024	65535	1025	3049
#3	170.210.0.1:80	4500	1024	65535	1026	6550
#4	170.210.0.1:80	4500	1024	65535	1027	6551
#5	128.0.0.1:80	1000	1024	65535	1028	3052

Table 2: Sample scenario for the novel port obfuscation algorithm

The first two entries of the table illustrate the contents of each of the variables when two ephemeral ports are selected to establish two consecutive connections to the same remote socket {128.0.0.1, 80}. We can see that the two ephemeral ports that get selected belong to the same port number “sequence”, as the result of the hash function F() is the same in these two cases.

The second and third entries of the table illustrate the contents of each of the variables when the algorithm later selects two ephemeral ports to establish two consecutive connections to the remote socket {172.0.0.1, 110}. We can see that the result of F() is the same for these two cases, and thus the two ephemeral ports that get selected belong to the same “sequence”. However, this sequence is different from that of the first two port numbers selected before, as the value of F() is different from the one obtained for those two ports numbers (#1 and #2) selected earlier.

Finally, when the algorithm later selects another ephemeral port to connect to the same socket as in #1 and #2, we note that the selected port number somehow belongs to the same sequence as the first two port numbers selected (#1 and #2), but that two ports of that sequence (3050 and 3051) have been skipped. This is the consequence of having a single global next_ephemeral variable that gets incremented whenever a port number is selected. When next_ephemeral is incremented as a result of the port selections #3 and #4, this causes two ports (3050 and 3051) in all the other the port number sequences (including that of #1 and #2) to be “skipped”, unnecessarily.

As in the case of the traditional TCP port selection algorithm, having a single global counter for the port numbers that have so far been selected may result in a port reuse frequency higher than needed.

The obvious mitigation for this effect would be to have a different “next_ephemeral” variable for each possible result of F(). Thus, assuming no hash collisions, the selection of a port number would increment only the corresponding next_ephemeral variable, without causing port numbers in other sequences to be skipped. However, this would likely require an unacceptable amount of memory.

A middle-ground between a single global next_ephemeral variable and a large number of next_ephemeral variables (one for each possible result of F()) would be to have a small number of next_ephemeral variables, such that each possible value of F() is matched to one of these variables. For example, we could define an array of 256 variables, each of them representing a different next_ephemeral variable. The index into this array could be the result of a hash function G() computed with the remote TCP socket {remote IP address, remote TCP port} and a secret key, or even a value derived from the result of F() (for example, the result of performing an eXclusive-OR among each of the bytes composing the result of F(). Figure 3 shows the pseudocode for the improved algorithm.

```
/* Initialization code */
for(i = 0; i < TABLE_LENGTH; i++)
    table[i] = random % 65536;


/* Ephemeral port selection */
offset = F(local_IP, remote_IP, remote_port, secret_key);
index = G(offset);
count = max_ephemeral - min_ephemeral + 1;

do {
    port = min_ephemeral + (offset + table[index])
               % (max_ephemeral - min_ephemeral + 1);

    table[index]++;
    count--;

      if(four-tuple is unique)
    return port;

} while (count > 0);

return ERROR;
```

Figure 4: An improvement to the novel port obfuscation algorithm

In order to illustrate how this slight modification can improve the port reuse properties of the novel port obfuscation algorithm discussed earlier, we can analyze the scenario of the previous section, this time from the perspective of the improved algorithm. For the purposes of illustrating how the improved algorithm works, we will refer to the array of next_ephemeral variables as “table”, and for simplicity-sake we will assume that all the entries of the array have been initialized to 1024. Also, we will use the variable “index” to store the value used as the index into the array “table”.

Table 3 illustrates a possible result for the same sequence of events as those in table 2, along with the values of each of the involved variables.

Nr.	IP:port	offset	min_ephemeral	max_ephemeral	index	table[index]	port
#1	128.0.0.1:80	1000	1024	65535	10	1024	3048
#2	128.0.0.1:80	1000	1024	65535	10	1025	3049
#3	170.210.0.1:80	4500	1024	65535	15	1024	6548
#4	170.210.0.1:80	4500	1024	65535	15	1025	6549
#5	128.0.0.1:80	1000	1024	65535	10	1026	3050

Table 3: Sample scenario for the improved novel port obfuscation algorithm

From the table we can see that the destinations “128.0.0.1:80” and “170.210.0.1:80” result in different values for “index” and, as a result, our slight modification successfully avoids the increments in one of the port number sequences to affect the other sequences, thus minimizing the port reuse frequency.

3.2.3 Secret Key

Every complex manipulation (like MD5) is no more secure than the input values, and in the case of ephemeral ports, the secret key. If an attacker is aware of which cryptographic hash function is being used by the victim (which we should expect), and the attacker can obtain enough material (e.g. ephemeral ports chosen by the victim), the attacker might simply search the entire secret key space to find matches.

To protect against this, the secret key should be of a reasonable length. Key-lengths of 32-bits should be adequate, since a 32-bit secret would result in approximately 65k possible secrets if the attacker is able to obtain a single ephemeral port (assuming a good hash function). If the attacker is able to obtain more ephemeral ports, key-lengths of 64-bits or more should be used.

Another possible mechanism for protecting the secret key is to change it after some time. If the host platform is capable of producing reasonable good random data, the secret key can be changed. Changing the secret will cause abrupt shifts in the chosen ephemeral ports, and consequently collisions may occur. Thus the change in secret key should be done with consideration and could be performed whenever one of the following events occur:

- Some predefined/random time has expired
- The secret has been used N times (i.e. we consider it insecure).
- There are few active connections (i.e., possibility of collision is low).
- There is little traffic (the performance overhead of collisions is tolerated).
- There is enough random data available to change the secret key (pseudo-random changes should not be done).

4 CONCLUSIONS

Ephemeral port randomization can provide efficient mitigation for blind attacks against transport protocols, without any key management overhead. Implementation and deployment experience have shown that trivial port randomization approaches can lead to interoperability problems. In this paper we have examined a novel algorithm to achieve port number obfuscation, while avoiding the potential interoperability problems introduced by other alternative approaches.

4 ACKNOWLEDGEMENTS

The author would like to thank Guillermo Gont and Juan Fraschini for reviewing a draft version of this paper. The author would also like to thank FreeBSD's Mike Silbersack for a very fruitful discussion about ephemeral port selection techniques

5 REFERENCES

[1] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[2] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.

[3] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[4] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[5] Bellovin, S., "Defending Against Sequence Number Attacks", RFC 1948, May 1996.

[6] Heffernan, A., "Protection of BGP Sessions via the TCP MD5 Signature Option", RFC 2385, August 1998.

[7] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.

[8] Stewart, R., Xie, Q., Morneault, K., Sharp, C., Schwarzbauer, H., Taylor, T., Rytina, I., Kalla, M., Zhang, L., and V. Paxson, "Stream Control Transmission Protocol", RFC 2960, October 2000.

[9] Kohler, E., Handley, M., and S. Floyd, "Datagram Congestion Control Protocol (DCCP)", RFC 4340, March 2006.

[10] Watson, P., "Slipping in the Window: TCP Reset attacks", December 2003.

[11] “IANA Port Numbers”, <http://www.iana.org/assignments/port-numbers>.

[12] Touch, J., "Defending TCP Against Spoofing Attacks", draft-ietf-tcpm-tcp-antispoof-05 (work in progress), October 2006.

[13] Rekhter, Y., Li, T., Hares, S., “A Border Gateway Protocol 4 (BGP-4)”, RFC 4271, January 2006.

[14] Larsen, M., Gont, F., "Port Randomization", draft-larsen-tsvwg-port-randomization-01.txt (work in progress), Feb. 2007.

[15] Gont, F., "ICMP attacks against TCP", draft-ietf-tcpm-icmp-attacks-01 (work in progress), October 2006.