Transport Area Working Group (tsvwg)
Internet-Draft
Intended status: BCP
Expires: August 19, 2010

M. Larsen
TietoEnator
F. Gont
UTN/FRH
February 15, 2010

Transport Protocol Port Randomization Recommendations
draft-ietf-tsvwg-port-randomization-06

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 five-tuple (Protocol, Source Address, Destination Address, Source Port, Destination Port) that identifies the transport protocol instance to be attacked. This document describes a number of simple and efficient methods for the selection of the client port number, such that the possibility of an attacker guessing the exact value is reduced. While this is not a replacement for cryptographic methods for protecting the transport-protocol instance, the described port number obfuscation algorithms provide improved security/obfuscation with very little effort and without any key management overhead. The algorithms described in this document are local policies that may be incrementally deployed, and that do not violate the specifications of any of the transport protocols that may benefit from them, such as TCP, UDP, UDP-lite, SCTP, DCCP, and RTP (provided the RTP application explicitly signals the RTP and RTCP port numbers).

http://www.ietf.org/ietf/1id-abstracts.txt.

The list of Internet-Draft Shadow Directories can be accessed at http://www.ietf.org/shadow.html.

This Internet-Draft will expire on August 19, 2010.

Copyright Notice

Table of Contents

1. Introduction

Recently, awareness has been raised about a number of "blind" attacks (i.e., attacks that can be performed without the need to sniff the packets that correspond to the transport protocol instance to be attacked) that can be performed against the Transmission Control Protocol (TCP) [RFC0793] and similar protocols. The consequences of these attacks range from throughput-reduction to broken connections or data corruption [I-D.ietf-tcpm-icmp-attacks] [RFC4953] [Watson].

All these attacks rely on the attacker's ability to guess or know the five-tuple (Protocol, Source Address, Source port, Destination Address, Destination Port) that identifies the transport protocol instance to be attacked.

Services are usually located at fixed, 'well-known' ports [IANA] at the host supplying the service (the server). Client applications connecting to any such service will contact the server by specifying the server IP address and service port number. The IP address and port number of the client are normally left unspecified by the client application and thus chosen automatically by the client networking stack. Ports chosen automatically by the networking stack are known as ephemeral ports [Stevens].

While the server IP address and well-known port and the client IP address may be known by an attacker, the ephemeral port of the client is usually unknown and must be guessed.

This document describes a number of algorithms for the selection of ephemeral port numbers, such that the possibility of an off-path attacker guessing the exact value is reduced. They are not a replacement for cryptographic methods of protecting a transport-protocol instance such as IPsec [RFC4301], the TCP MD5 signature option [RFC2385], or the TCP Authentication Option [I-D.ietf-tcpm-tcp-auth-opt]. For example, they do not provide any mitigation in those scenarios in which the attacker is able to sniff the packets that correspond to the transport protocol instance to be attacked. However, the proposed algorithms provide improved obfuscation with very little effort and without any key management overhead.

The mechanisms described in this document are local modifications that may be incrementally deployed, and that do not violate the specifications of any of the transport protocols that may benefit from them, such as TCP [RFC0793], UDP [RFC0768], SCTP [RFC4960], DCCP [RFC4340], UDP-lite [RFC3828], and RTP [RFC3550] (provided the RTP application explicitly signals the RTP and RTCP port numbers with e.g.[RFC3605]).

Since these mechanisms are obfuscation techniques, focus has been on a reasonable compromise between the level of obfuscation and the ease of implementation. Thus the algorithms must be computationally efficient, and not require substantial state.

We note that while the technique of mitigating "blind" attacks by obfuscating the ephemeral port selection is well-known as "port randomization", the goal of the algorithms described in this document is to reduce the chances of an attacker guessing the ephemeral ports selected for new transport protocol instances, rather than to actually produce mathematically random sequences of ephemeral ports.

Throughout this document we will use the term "transport-protocol instance" as a general term to refer to an instantiation of a transport protocol (e.g, a "connection" in the case of connection-oriented transport protocols) and the term "instance-id" as a short-handle to refer to the group of values that identify a transport-protocol instance (e.g., in the case of TCP, the five-tuple {Protocol, IP Source Address, TCP Source Port, IP Destination Address, TCP Destination Port}).

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. Ephemeral Ports

2.1. Traditional Ephemeral Port 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 [IANA]. IANA has 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 dynamic port range defined by IANA consists of the 49152-65535 range, and is meant for the selection of ephemeral ports.

2.2. Ephemeral port selection

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

Selection of ephemeral ports such that they result in unique instance-id's (five-tuples) is handled by some implementations by having a per-protocol global 'next_ephemeral' variable that is equal to the previously chosen ephemeral port + 1, i.e. the selection process is:

```
/* Initialization at system boot time. Could be random */
next_ephemeral = min_ephemeral;

/* Ephemeral port selection function */
count = max_ephemeral - min_ephemeral + 1;

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

    if (five-tuple is unique)
        return port;

    count--;

} while (count > 0);

return ERROR;
```

Figure 1

This algorithm works adequately provided that the number of transport-protocol instances (for a each transport protocol) that have a life-time longer than it takes to exhaust the total ephemeral port range is small, so that collisions of instance-id's are rare.

However, this method has the drawback that the 'next_ephemeral' variable and thus the ephemeral port range is shared between all transport-protocol instances and the next ports chosen by the client are easy to predict. If an attacker operates an "innocent" server to which the client connects, it is easy to obtain a reference point for the current value of the 'next_ephemeral' variable. Additionally, if an attacker could force a client to periodically establish e.g., a new TCP connection to an attacker controlled machine (or through an attacker observable routing path), the attacker could subtract consecutive source port values to obtain the number of outgoing TCP connections established globally by the target host within that time period (up to wrap-around issues and instance-id collisions, of course).

2.3. Collision of instance-id's

While it is possible for the ephemeral port selection algorithm to verify that the selected port number results in a instance-id that is

not currently in use by that system, the resulting instance-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 instance-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 instance-id as the one in the TIME_WAIT state at the server), an instance-id "collision" would occur. The effect of these collisions range from connection-establishment failures to TIME-WAIT state assassination (with the potential of data corruption) [RFC1337]. 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 minimize the rate of instance-id collisions.

A simple approach to minimize the rate of these collisions would be to choose port numbers incrementally, so that a given port number would not be reused until the rest of the port numbers in ephemeral port range have been used for a transport protocol instance. However, if a single global variable were used to keep track of the last ephemeral port selected, ephemeral port numbers would be trivially predictable, thus making it easier for an off-path attacker to "guess" the instance-id in use by a target transport-protocol instance. Section 3.3.3 and Section 3.3.4 describe algorithms that select port numbers incrementally, while still making it difficult for an off-path attacker to predict the ephemeral ports used for future transport-protocol instances.

A simple but inefficient approach to minimize the rate of collisions of instance-id's would be, e.g. in the case of TCP, for both end-points of a TCP connection to keep state about recent connections (e.g., have both end-points end up in the TIME-WAIT state).

3. Obfuscating the Ephemeral Ports

3.1. Characteristics of a good ephemeral port obfuscation algorithm

There are a number of factors to consider when designing an algorithm for selecting ephemeral ports, which include:

- Minimizing the predictability of the ephemeral port numbers used for future transport-protocol instances.
- Minimizing collisions of instance-id's
- Avoiding conflict with applications that depend on the use of specific port numbers.

Given the goal of improving the transport protocol's resistance to attack by obfuscation of the instance-id, it is key to minimize the predictability of the ephemeral ports that will be selected for new transport-protocol instances. While the obvious approach to address this requirement would be to select the ephemeral ports by simply picking a random value within the chosen port number range, this straightforward policy may lead to collisions of instance-id's, which could lead to the interoperability problems (e.g., delays in the establishment of new connections, failures in connection-establishment, or data corruption) discussed in Section 2.3. As discussed in Section 1, it is worth noting that while the technique of mitigating "blind" attacks by obfuscating the ephemeral port election is well-known as "port randomization", the goal of the algorithms described in this document is to reduce the chances of an attacker guessing the ephemeral ports selected for new transport-protocol instances, rather than to actually produce sequences of mathematically random ephemeral port numbers.

It is also worth noting that, provided adequate algorithms are in use, the larger the range from which ephemeral pots are selected, the smaller the chances of an attacker are to guess the selected port number.

In scenarios in which a specific client establishes transport-protocol instances with a specific service at a server, the problems described in Section 2.3 become evident. A good algorithm to minimize the collisions of instance-id's would consider the time a given five-tuple was last used, and would avoid reusing the last recently used five-tuples. A simple approach to minimize the rate of collisions would be to choose port numbers incrementally, so that a given port number would not be reused until the rest of the port numbers in the ephemeral port range have been used for a transport protocol instance. However, if a single global variable were used to

keep track of the last ephemeral port selected, ephemeral port numbers would be trivially predictable.

It is important to note that a number of applications rely on binding specific port numbers that may be within the ephemeral ports range. If such an application was run while the corresponding port number was in use, the application would fail. Therefore, ephemeral port selection algorithms avoid using those port numbers.

Port numbers that are currently in use by a TCP in the LISTEN state should not be allowed for use as ephemeral ports. If this rule is not complied with, an attacker could potentially "steal" an incoming connection to a local server application by issuing a connection request to the victim client at roughly the same time the client tries to connect to the victim server application [CPNI-TCP] [I-D.gont-tcp-security]. If the SYN segment corresponding to the attacker's connection request and the SYN segment corresponding to the victim client "cross each other in the network", and provided the attacker is able to know or guess the ephemeral port used by the client, a TCP simultaneous open scenario would take place, and the incoming connection request sent by the client would be matched with the attacker's socket rather than with the victim server application's socket.

It should be noted that most applications based on popular implementations of the TCP API (such as the Sockets API) perform "passive opens" in three steps. Firstly, the application obtains a file descriptor to be used for inter-process communication (e.g., by issuing a socket() call). Secondly, the application binds the file descriptor to a local TCP port number (e.g., by issuing a bind() call), thus creating a TCP in the fictional CLOSED state. Thirdly, the aforementioned TCP is put in the LISTEN state (e.g., by issuing a listen() call). As a result, with such an implementation of the TCP API, even if port numbers in use for TCPs in the LISTEN state were not allowed for use as ephemeral ports, there is a window of time between the second and the third steps in which an attacker could be allowed to select a port number that would be later used for listening to incoming connections. Therefore, these implementations of the TCP API should enforce a stricter requirement for the allocation of port numbers: port numbers that are in use by a TCP in the LISTEN or CLOSED states should not be allowed for allocation as ephemeral ports [CPNI-TCP] [I-D.gont-tcp-security].

The aforementioned issues do not affect SCTP, since most SCTP implementations do not allow a socket to be bound to the same port number unless a specific socket option (SCTP_REUSE_PORT) is issued on the socket (i.e., this behavior needs to be explititly allowed beforehand). An example of a typical SCTP socket API can be found in

[I-D.ietf-tsvwg-sctpsocket].

DCCP is not affected is not affected by the exploitation of "simultaneous opens" to ""steal" incoming connections, as the server and the client state machines are different [RFC4340]. However, it may be affected by the vector involving binding a more specific socket. As a result, those tuples {local IP address, local port, Service Code} that are in use by a local socket should not be allowed for allocation as ephemeral ports.

3.2. Ephemeral port number range

As mentioned in Section 2.1, the dynamic ports consist of the range 49152-65535. However, ephemeral port selection algorithms should use the whole range 1024-49151.

Since this range includes ports numbers assigned by IANA, this may not always be possible, though. A possible workaround for this potential problem would be to maintain a local list of the port numbers that should not be allocated as ephemeral ports. Thus, before allocating a port number, the ephemeral port selection function would check this list, avoiding the allocation of ports that may be needed for specific applications.

Ephemeral port selection algorithms SHOULD use the largest possible port range, since this improves obfuscation.

3.3. Ephemeral Port Obfuscation Algorithms

Ephemeral port selection algorithms SHOULD obfuscate the allocation of their ephemeral ports, since this helps to mitigate a number of attacks that depend on the attacker's ability to guess or know the five-tuple that identifies the transport protocol instance to be attacked.

The following subsections describe a number of algorithms that could be implemented in order to obfuscate the selection of ephemeral port numbers.

3.3.1. Algorithm 1: Simple port randomization algorithm

In order to address the security issues discussed in Section 1 and Section 2.2, a number of systems have implemented simple ephemeral port number randomization, as follows:

```
/* Ephemeral port selection function */
num_ephemeral = max_ephemeral - min_ephemeral + 1;
next_ephemeral = min_ephemeral + (random() % num_ephemeral);
count = num_ephemeral;

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

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

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

return ERROR;
```

Figure 2

We will refer to this algorithm as 'Algorithm 1'.

Note: "random()" is a function that returns a pseudo-random unsigned interger number in the range 0-65535 (it may return values larger than 65535, as is the case with the "random()" C-language function).

Since the initially chosen port may already be in use with identical IP addresses and server port, the resulting five-tuple might not be unique. Therefore, multiple ports may have to be tried and verified against all existing transport-protocol instances before a port can be chosen.

Web proxy servers, NAPTs [RFC2663], and other middle-boxes aggregate multiple peers into the same port space and thus increase the population of used ephemeral ports, and hence the chances of collisions of instance-id's. However, [Allman] has shown that at least in the network scenarios used for measuring the collision properties of the algorithms described in this document, the collision rate resulting from the use of the aforementioned middle-boxes is nevertheless very low.

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, thus possibly leading to collisions of instance-id's. Even if a given five-tuple is verified to be unique by the port selection

algorithm, the five-tuple might still be in use at the remote system. In such a scenario, a connection request could possibly fail ([Silbersack] describes this problem for the TCP case).

This algorithm selects ephemeral port numbers randomly and thus reduces the chances of an attacker of guessing the ephemeral port selected for a target transport-protocol instance. Additionally, it prevents attackers from obtaining the number of outgoing transport-protocol instances (e.g., TCP connections) established by the client in some period of time.

3.3.2. Algorithm 2: Another simple port randomization algorithm

The following pseudo-code illustrates another algorithm for selecting a random port number, in which in the event a local instance-id collision is detected, another port number is selected randomly:

```
/* Ephemeral port selection function */
num_ephemeral = max_ephemeral - min_ephemeral + 1;
next_ephemeral = min_ephemeral + (random() % num_ephemeral);
count = num_ephemeral;

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

    next_ephemeral = min_ephemeral + (random() % num_ephemeral);
    count--;
} while (count > 0);

return ERROR;
```

Figure 3

We will refer to this algorithm as 'Algorithm 2'. This algorithm might be unable to select an ephemeral port (i.e., return "ERROR") even if there are port numbers that would result in unique five-tuples, when there are a large number of port numbers already in use. However, the results in [Allman] have shown that in common scenarios, one port choice is enough, and in most cases where more than one choice is needed two choices suffice. Therefore, in those scenarios this would not be problem.

3.3.3. Algorithm 3: Simple hash-based algorithm

We would like to achieve the port reuse properties of the traditional BSD port selection algorithm (described in Section 2.2), while at the

same time achieve the obfuscation properties of Algorithm 1 and Algorithm 2.

Ideally, we would like a 'next_ephemeral' value for each set of (local IP address, remote IP addresses, remote port), so that the port reuse frequency is the lowest possible. Each of these 'next_ephemeral' variables should be initialized with random values within the ephemeral port range and would thus separate the ephemeral port space of the transport-protocol instances on a "per destination end-point" basis (this "separation of the ephemeral port space" means that transport-protocol instances with different remote end-points will not have different sequences of port numbers; i.e., wil not be part of the same ephemeral port sequence as in the case of the traditional BSD ephemeral port selection algorithm). Since we do not want to maintain in memory all these 'next_ephemeral' values, we propose an offset function F(), that can be computed from the local IP address, remote IP address, remote port and a secret key. F() will yield (practically) different values for each set of arguments, i.e.:

```
/* Initialization at system boot time. Could be random. */
next_ephemeral = 0;

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

do {
    port = min_ephemeral +
           (next_ephemeral + offset) % num_ephemeral;

    next_ephemeral++;

    if(resulting five-tuple is unique)
        return port;

    count--;

} while (count > 0);

return ERROR;
```

Figure 4

We will refer to this algorithm as 'Algorithm 3'.

In other words, the function F() provides a "per destination end-point" fixed offset within the global ephemeral port range. Both the 'offset' and 'next_ephemeral' variables may take any value within the storage type range since we are restricting the resulting port in a similar way as in the Algorithm 1 (described in Section 3.3.1). This allows us to simply increment the 'next_ephemeral' variable and rely on the unsigned integer to simply wrap-around.

The function F() should be a cryptographic hash function like MD5 [RFC1321]. The function should use both IP addresses, the remote port and a secret key value to compute the offset. The remote IP address is the primary separator and must be included in the offset calculation. The local IP address and remote port may in some cases be constant and not improve the ephemeral port space separation, however, they should also be included in the offset calculation.

Cryptographic algorithms stronger than e.g. MD5 should not be necessary, given that Algorithm #3 is simply an obfuscation technique. The secret should be chosen as random as possible, see [RFC4086] for recommendations on choosing secrets.

Note that on multiuser systems, the function F() could include user specific information, thereby providing protection not only on a host to host basis, but on a user to service basis. In fact, any identifier of the remote entity could be used, depending on availability an the granularity requested. With SCTP both hostnames and alternative IP addresses may be included in the association negotiation and either of these could be used in the offset function F().

When multiple unique identifiers are available, any of these can be chosen as input to the offset function F() since they all uniquely identify the remote entity. However, in cases like SCTP where the ephemeral port must be unique across all IP address permutations, we should ideally always use the same IP address to get a single starting offset for each association negotiation from a given remote entity to minimize the possibility of collisions. A simple numerical sorting of the IP addresses and always using the numerically lowest could achieve this. However, since most protocols most likely will report the same IP addresses in the same order in each association setup, this sorting is most likely not necessary and the 'first one' can simply be used.

The ability of hostnames to uniquely define hosts can be discussed, and since SCTP always includes at least one IP address, we recommend to use this as input to the offset function F() and ignore hostnames chunks when searching for ephemeral ports.

It should be noted that, as this algorithm uses a global counter ("next_ephemeral") for selecting ephemeral ports, if an attacker could e.g., force a client to periodically establish a new TCP connections to an attacker controlled machine (or through an attacker observable routing path), the attacker could subtract consecutive source port values to obtain the number of outgoing TCP connections established globally by the target host within that time period (up to wrap-around issues and 5-tuple collisions, of course).

3.3.4. Algorithm 4: Double-hash obfuscation algorithm

A tradeoff between maintaining a single global 'next_ephemeral' variable and maintaining 2**N 'next_ephemeral' variables (where N is the width of the result of F()) could be achieved as follows. The system would keep an array of TABLE_LENGTH short integers, which would provide a separation of the increment of the 'next_ephemeral' variable. This improvement could be incorporated into Algorithm 3 as follows:

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


/* Ephemeral port selection function */
num_ephemeral = max_ephemeral - min_ephemeral + 1;
offset = F(local_IP, remote_IP, remote_port, secret_key1);
index = G(local_IP, remote_IP, remote_port, secret_key2);
count = num_ephemeral;

do {
    port = min_ephemeral + (offset + table[index]) % num_ephemeral;
    table[index]++;

    if(resulting five-tuple is unique)
        return port;

   count--;

} while (count > 0);

return ERROR;
```

Figure 5

We will refer to this algorithm as 'Algorithm 4'.

'table[]' could be initialized with mathematically random values, as indicated by the initialization code in pseudo-code above. The function G() should be a cryptographic hash function like MD5 [RFC1321]. It should use both IP addresses, the remote port and a secret key value to compute a value between 0 and (TABLE_LENGTH-1). Alternatively, G() could take as "offset" as input, and perform the exclusive-or (xor) operation between all the bytes in 'offset'.

The array 'table[]' assures that successive transport-protocol instances with the same remote end-point will use increasing ephemeral port numbers. However, incrementation of the port numbers is separated into TABLE_LENGTH different spaces, and thus the port reuse frequency will be (probabilistically) lower than that of Algorithm 3. That is, a new tranport-protocol instance with some remote end-point will not necessarily cause the 'next_ephemeral' variable corresponding to other end-points to be incremented.

It is interesting to note that the size of 'table[]' does not limit the number of different port sequences, but rather separates the *increments* into TABLE_LENGTH different spaces. The port sequence will result from adding the corresponding entry of 'table[]' to the variable 'offset', which selects the actual port sequence (as in Algorithm 3). [Allman] has found that a TABLE_LENGTH of 10 can result in an improvement over Algorithm 3. Further increasing the TABLE_LENGTH will increase the obfuscation, and possibly further decrease the collision rate.

An attacker can perform traffic analysis for any "increment space" into which the attacker has "visibility", namely that the attacker can force the client to establish a transport-protocol instance whose G(offset) identifies the target "increment space". However, the attacker's ability to perform traffic analysis is very reduced when compared to the traditional BSD algorithm (described in Section 2.2) and Algorithm 3. Additionally, an implementation can further limit the attacker's ability to perform traffic analysis by further separating the increment space (that is, using a larger value for TABLE_LENGTH).

3.3.5. Algorithm 5: Random-increments port selection algorithm

[Allman] introduced another port obfuscation algorithm, which offers a middle ground between the algorithms that select ephemeral ports randomly (such as those described in Section 3.3.1 and Section 3.3.2), and those that offer obfuscation but no randomization (such as those described in Section 3.3.3 and Section 3.3.4). We will refer to this algorithm as 'Algorithm 5'.

```
/* Initialization code at system boot time. */
next_ephemeral = random() % 65536;   /* Initialization value */
N = 500;                             /* Determines the tradeoff */

/* Ephemeral port selection function */
num_ephemeral = max_ephemeral - min_ephemeral + 1;

count = num_ephemeral;

do {
    next_ephemeral = next_ephemeral + (random() % N) + 1;
    port = min_ephemeral + (next_ephemeral % num_ephemeral);

    if(resulting five-tuple is unique)
            return port;

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

return ERROR;
```

Figure 6

This algorithm aims at at producing a monotonically-increasing sequence to prevent the collision of instance-id's, while avoiding the use of fixed increments, which would lead to trivially-predictable sequences. The value "N" allows for direct control of the tradeoff between the level of obfuscation and the port reuse frequency. The smaller the value of "N", the more linear the more similar this algorithm is to the traditional BSD port selection algorithm (described in Section 2.2. The larger the value of "N", the more similar this algorithm is to the algorithm described in Section 3.3.1 of this document.

When the port numbers wrap, there is the risk of collisions of instance-id's. Therefore, "N" should be selecting according to the following criteria:

- It should maximize the wrapping time of the ephemeral port space
- It should minimize collisions of instance-id's
- It should maximize obfuscation

Clearly, these are competing goals, and the decision of which value of "N" to use is a tradeoff. Therefore, the value of "N" should be configurable so that system administrators can make the tradeoff for themselves.

3.4. Secret-key considerations for hash-based port obfuscation algorithms

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 may 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 automatically.

Changing the secret will cause abrupt shifts in the chosen ephemeral ports, and consequently collisions may occur. That is, upon changing the secret, the "offset" value (see Section 3.3.3 and Section 3.3.4) used for each destination end-point will be different from that computed with the previous secret, thus leading to the selection of a port number recently used for connecting to the same end-point.

Thus the change in secret key should be done with consideration and could be performed whenever one of the following events occur:

- The system is being bootstrapped.
- Some predefined/random time has expired.
- The secret has been used N times (i.e. we consider it insecure).
- There are few active transport protocol instances (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).

3.5. Choosing an ephemeral port obfuscation algorithm

[Allman] is an empirical study of the properties of the algorithms described in this document, which has found that all the algorithms described in this document offer low collision rates -- at most 0.3%. That is, in those network scenarios assessed by [Allman] all of the algorithms described in this document perform well in terms of collisions of instance-id's. However, these results may vary depending on the characteristics of network traffic and the specific network setup.

The algorithm described in Section 2.2 is the traditional ephemeral port selection algorithm implemented in BSD-derived systems. It generates a global sequence of ephemeral port numbers, which makes it trivial for an attacker to predict the port number that will be used for a future transport protocol instance. However, it is very simple, and leads to a low port reuse frequency.

Algorithm 1 and Algorithm 2 have the advantage that they provide actual randomization of the ephemeral ports. However, they may increase the chances of port number collisions, which could lead to the failure of a connection establishment attempt. [Allman] found that these two algorithms show the largest collision rates (among all the algorithms described in this document).

Algorithm 3 provides complete separation in local and remote IP addresses and remote port space, and only limited separation in other dimensions (see Section 3.4). However, implementations should consider the performance impact of computing the cryptographic hash used for the offset.

Algorithm 4 improves Algorithm 3, usually leading to a lower port reuse frequency, at the expense of more processor cycles used for computing G(), and additional kernel memory for storing the array 'table[]'.

Algorithm 5 offers middle ground between the simple randomization algorithms (Algorithm 1 and Algorithm 2) and the hash-based algorithms (Algorithm 3 and Algorithm 4). The upper limit on the random increments (the value "N" in the pseudo-code included in Section 3.3.5 controls the trade-off between randomization and port-reuse frequency.

Finally, a special case that may preclude the utilization of Algorithm 3 and Algorithm 4 should be analyzed. There exist some applications that contain the following code sequence:

```
s = socket();
bind(s, IP_address, port = *);
```

Figure 7

In some BSD-derived systems, the call to bind() will result in the selection of an ephemeral port number. However, as neither the remote IP address nor the remote port will be available to the ephemeral port selection function, the hash function F() used in Algorithm 3 and Algorithm 4 will not have all the required arguments, and thus the result of the hash function will be impossible to compute. Transport protocols implementing Algorithm 3 or Algorithm 4 should consider using Algorithm 2 when facing the scenario just described.

An alternative to this behavior would be to implement "lazy binding" in response to the bind() call. That is, selection of an ephemeral port would be delayed until, e.g., connect() or send() are called. Thus, at that point the ephemeral port is actually selected, all the necessary arguments for the hash function F() would be available, and thus Algorithm 3 and Algorithm 4 could still be used in this scenario. This algorithm has been implemented by Linux [Linux].

4. Port obfuscation and Network Address Port Translation (NAPT)

Network Address Port Translation (NAPT) translate both the network address and transport-protocol port number, thus allowing the transport identifiers of a number of private hosts to be multiplexed into the transport identifiers of a single external address. [RFC2663]

In those scenarios in which a NAPT is present between the two end-points of transport-protocol instance, the obfuscation of the ephemeral ports (from the point of view of the external network) will depend on the ephemeral port selection function at the NAPT. Therefore, NAPTs should consider obfuscating the ephemeral ports by means of any of the algorithms discussed in this document. It should be noted that in some network scenarios, a NAPT may naturally obscure ephemeral port selections simply due to the vast range of services with which it establishes connections and to the overall rate of the traffic [Allman].

Section 3.5 provides guidance in choosing a port obfuscation algorithm.

5. Security Considerations

Obfuscating ephemeral ports is no replacement for cryptographic mechanisms, such as IPsec [RFC4301], in terms of protecting transport-protocol instances against blind attacks.

An eavesdropper, which can monitor the packets that correspond to the transport-protocol instance to be attacked could learn the IP addresses and port numbers in use (and also sequence numbers etc.) and easily perform an attack. Ephemeral port obfuscation does not provide any additional protection against this kind of attacks. In such situations, proper authentication mechanisms such as those described in [RFC4301] should be used.

If the local offset function F() results in identical offsets for different inputs, the port-offset mechanism proposed in this document has no or reduced effect.

If random numbers are used as the only source of the secret key, they must be chosen in accordance with the recommendations given in [RFC4086].

If an attacker uses dynamically assigned IP addresses, the current ephemeral port offset (Algorithm 3 and Algorithm 4) for a given five-tuple can be sampled and subsequently used to attack an innocent peer reusing this address. However, this is only possible until a re-keying happens as described above. Also, since ephemeral ports are only used on the client side (e.g. the one initiating the transport-protocol communication), both the attacker and the new peer need to act as servers in the scenario just described. While servers using dynamic IP addresses exist, they are not very common and with an appropriate re-keying mechanism the effect of this attack is limited.

6. IANA Considerations

There are no IANA registries within this document. The RFC-Editor can remove this section before publication of this document as an RFC.

7. Acknowledgements

The offset function was inspired by the mechanism proposed by Steven Bellovin in [RFC1948] for defending against TCP sequence number attacks.

The authors would like to thank (in alphabetical order) Mark Allman, Matthias Bethke, Stephane Bortzmeyer, Brian Carpenter, Vincent Deffontaines, Lars Eggert, Gorry Fairhurst, Guillermo Gont, Alfred Hoenes, Amit Klein, Carlos Pignataro, Kacheong Poon, Pasi Sarolahti, Randall Stewart, Joe Touch, Michael Tuexen, and Dan Wing for their valuable feedback on earlier versions of this document.

The authors would like to thank FreeBSD's Mike Silbersack for a very fruitful discussion about ephemeral port selection techniques.

Fernando Gont would like to thank Carolina Suarez for her love and support.

8. References

8.1. Normative References

[RFC0768] Postel, J., "User Datagram Protocol", STD 6, RFC 768, August 1980.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, September 1981.

[RFC1321] Rivest, R., "The MD5 Message-Digest Algorithm", RFC 1321, April 1992.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC2385] Heffernan, A., "Protection of BGP Sessions via the TCP MD5 Signature Option", RFC 2385, August 1998.

[RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", STD 64, RFC 3550, July 2003.

[RFC3605] Huitema, C., "Real Time Control Protocol (RTCP) attribute in Session Description Protocol (SDP)", RFC 3605, October 2003.

[RFC3828] Larzon, L-A., Degermark, M., Pink, S., Jonsson, L-E., and G. Fairhurst, "The Lightweight User Datagram Protocol (UDP-Lite)", RFC 3828, July 2004.

[RFC4086] Eastlake, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, June 2005.

[RFC4301] Kent, S. and K. Seo, "Security Architecture for the Internet Protocol", RFC 4301, December 2005.

[RFC4340] Kohler, E., Handley, M., and S. Floyd, "Datagram Congestion Control Protocol (DCCP)", RFC 4340, March 2006.

[RFC4960] Stewart, R., "Stream Control Transmission Protocol", RFC 4960, September 2007.

8.2. Informative References

[FreeBSD] The FreeBSD Project, "http://www.freebsd.org".

[IANA] "IANA Port Numbers",

<http://www.iana.org/assignments/port-numbers>.

[I-D.ietf-tcpm-icmp-attacks]
Gont, F., "ICMP attacks against TCP", draft-ietf-tcpm-icmp-attacks-10 (work in progress), January 2010.

[RFC1337] Braden, B., "TIME-WAIT Assassination Hazards in TCP", RFC 1337, May 1992.

[RFC1948] Bellovin, S., "Defending Against Sequence Number Attacks", RFC 1948, May 1996.

[RFC2663] Srisuresh, P. and M. Holdrege, "IP Network Address Translator (NAT) Terminology and Considerations", RFC 2663, August 1999.

[RFC4953] Touch, J., "Defending TCP Against Spoofing Attacks", RFC 4953, July 2007.

[I-D.ietf-tsvwg-sctpsocket]
Stewart, R., Poon, K., Tuexen, M., Yasevich, V., and P. Lei, "Sockets API Extensions for Stream Control Transmission Protocol (SCTP)", draft-ietf-tsvwg-sctpsocket-21 (work in progress), February 2010.

[Allman] Allman, M., "Comments On Selecting Ephemeral Ports", ACM Computer Communication Review, 39(2), 2009.

[CPNI-TCP]
Gont, F., "CPNI Technical Note 3/2009: Security Assessment of the Transmission Control Protocol (TCP)", UK Centre for the Protection of National Infrastructure, 2009.

[I-D.gont-tcp-security]
Gont, F., "Security Assessment of the Transmission Control Protocol (TCP)", draft-gont-tcp-security-00 (work in progress), February 2009.

[Linux] The Linux Project, "http://www.kernel.org".

[NetBSD] The NetBSD Project, "http://www.netbsd.org".

[OpenBSD] The OpenBSD Project, "http://www.openbsd.org".

[OpenSolaris]
OpenSolaris, "http://www.opensolaris.org".

[Silbersack]
Silbersack, M., "Improving TCP/IP security through randomization without sacrificing interoperability.", EuroBSDCon 2005 Conference .

[Stevens] Stevens, W., "Unix Network Programming, Volume 1: Networking APIs: Socket and XTI", Prentice Hall , 1998.

[I-D.ietf-tcpm-tcp-auth-opt]
Touch, J., Mankin, A., and R. Bonica, "The TCP Authentication Option", draft-ietf-tcpm-tcp-auth-opt-10 (work in progress), January 2010.

[Watson] Watson, P., "Slipping in the Window: TCP Reset Attacks", CanSecWest 2004 Conference .

Appendix A. Survey of the algorithms in use by some popular implementations

A.1. FreeBSD

FreeBSD 8.0 implements Algorithm 1, and in response to this document now uses a 'min_port' of 10000 and a 'max_port' of 65535. [FreeBSD]

A.2. Linux

Linux 2.6.15-53-386 implements Algorithm 3. If the algorithm is faced with the corner-case scenario described in Section 3.5, Algorithm 1 is used instead [Linux].

A.3. NetBSD

NetBSD 5.0.1 does not obfuscate its ephemeral port numbers. It selects ephemeral port numbers from the range 49152-65535, starting from port 65535, and decreasing the port number for each ephemeral port number selected [NetBSD].

A.4. OpenBSD

OpenBSD 4.2 implements Algorithm 1, with a 'min_port' of 1024 and a 'max_port' of 49151. [OpenBSD]

A.5. OpenSolaris

OpenSolaris 2009.06 implements Algorithm 1, with a 'min_port' of 32768 and a 'max_port' of 65535. [OpenSolaris]

Appendix B. Changes from previous versions of the draft (to be removed by the RFC Editor before publication of this document as a RFC

B.1. Changes from draft-ietf-tsvwg-port-randomization-05

- Addresses AD review feedback from Lars Eggert.

B.2. Changes from draft-ietf-tsvwg-port-randomization-04

- Fixes nits.

B.3. Changes from draft-ietf-tsvwg-port-randomization-03

- Addresses WGLC comments from Mark Allman. See: http://www.ietf.org/mail-archive/web/tsvwg/current/msg09149.html

B.4. Changes from draft-ietf-tsvwg-port-randomization-02

- Added clarification of what we mean by "port randomization".
- Addresses feedback sent on-list and off-list by Mark Allman.
- Added references to [Allman] and [CPNI-TCP].

B.5. Changes from draft-ietf-tsvwg-port-randomization-01

- Added Section 2.3.
- Added discussion of "lazy binding in Section 3.5.
- Added discussion of obtaining the number of outgoing connections.
- Miscellaneous editorial changes

B.6. Changes from draft-ietf-tsvwg-port-randomization-00

- Added Section 3.1.
- Changed Intended Status from "Standards Track" to "BCP".
- Miscellaneous editorial changes.

B.7. Changes from draft-larsen-tsvwg-port-randomization-02

- Draft resubmitted as draft-ietf.

- Included references and text on protocols other than TCP.
- Added the second variant of the simple port randomization algorithm
- Reorganized the algorithms into different sections
- Miscellaneous editorial changes.

B.8. Changes from draft-larsen-tsvwg-port-randomization-01

- No changes. Draft resubmitted after expiration.

B.9. Changes from draft-larsen-tsvwg-port-randomization-00

- Fixed a bug in expressions used to calculate number of ephemeral ports
- Added a survey of the algorithms in use by popular TCP implementations
- The whole document was reorganized
- Miscellaneous editorial changes

B.10. Changes from draft-larsen-tsvwg-port-randomisation-00

- Document resubmitted after original document by M. Larsen expired in 2004
- References were included to current WG documents of the TCPM WG
- The document was made more general, to apply to all transport protocols
- Miscellaneous editorial changes

Authors' Addresses

Michael Vittrup Larsen
TietoEnator
Skanderborgvej 232
Aarhus DK-8260
Denmark

Phone: +45 8938 5100
Email: michael.larsen@tietoenator.com

Fernando Gont
Universidad Tecnologica Nacional / Facultad Regional Haedo
Evaristo Carriego 2644
Haedo, Provincia de Buenos Aires 1706
Argentina

Phone: +54 11 4650 8472
Email: fernando@gont.com.ar