

tswg
Internet-Draft
Intended status: Best Current
Practice
Expires: August 28, 2008

M. Larsen
TietoEnator
F. Gont
UTN/FRH
February 25, 2008

Port Randomization
draft-ietf-tswg-port-randomization-01

Status of this Memo

By submitting this Internet-Draft, each author represents that any applicable patent or other IPR claims of which he or she is aware have been or will be disclosed, and any of which he or she becomes aware will be disclosed, in accordance with [Section 6 of BCP 79](#).

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF), its areas, and its working groups. Note that other groups may also distribute working documents as Internet-Drafts.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

The list of current Internet-Drafts can be accessed at <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 28, 2008.

Copyright Notice

Copyright (C) The IETF Trust (2008).

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 simple and efficient method for random 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, the described port number randomization algorithms provide improved security/obfuscation with very little effort and without any key management overhead. The mechanisms described in this document are a local modification that may be incrementally deployed, and that does not violate the specifications of any of the transport protocols that may benefit from it, such as TCP, UDP, SCTP, DCCP, and RTP.

Table of Contents

1.	Introduction	4
2.	Ephemeral Ports	6
2.1.	Traditional Ephemeral Port Range	6
2.2.	Ephemeral port selection	6
3.	Randomizing the Ephemeral Ports	8
3.1.	Characteristics of a good ephemeral port randomization algorithm	8
3.2.	Ephemeral port number range	9
3.3.	Ephemeral Port Randomization Algorithms	9
3.3.1.	Algorithm 1: Simple port randomization algorithm	9
3.3.2.	Algorithm 2: Another simple port randomization algorithm	11
3.3.3.	Algorithm 3: Simple hash-based algorithm	11
3.3.4.	Algorithm 4: Double-hash randomization algorithm	13
3.4.	Secret-key considerations for hash-based port randomization algorithms	15
3.5.	Choosing an ephemeral port randomization algorithm	16
4.	Security Considerations	17
5.	Acknowledgements	18
6.	References	19
6.1.	Normative References	19
6.2.	Informative References	20
Appendix A.	Survey of the algorithms in use by some popular implementations	21
A.1.	FreeBSD	21
A.2.	Linux	21
A.3.	NetBSD	21
A.4.	OpenBSD	21
Appendix B.	Changes from previous versions of the draft	22
B.1.	Changes from draft-ietf-tsvwg-port-randomisation-00	22
B.2.	Changes from draft-larsen-tsvwg-port-randomisation-02	22
B.3.	Changes from draft-larsen-tsvwg-port-randomisation-01	22
B.4.	Changes from draft-larsen-tsvwg-port-randomization-00	22
B.5.	Changes from draft-larsen-tsvwg-port-randomisation-00	22
	Authors' Addresses	24
	Intellectual Property and Copyright Statements	25

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 accurately guessed by an attacker, the ephemeral port of the client is usually unknown and must be guessed.

This document describes a number of algorithms for random selection of the client ephemeral port, that reduce the possibility of an off-path attacker guessing the exact value. They are not a replacement for cryptographic methods of protecting a connection 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 connection 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 does not violate the specifications of any of the transport protocols that may benefit from it, such as TCP [[RFC0793](#)], UDP [[RFC0768](#)], SCTP [[RFC4960](#)], DCCP [[RFC4340](#)], UDP-lite [[RFC3828](#)], and RTP [[RFC3550](#)].

Since these mechanisms are obfuscation techniques, focus has been on a reasonable compromise between level of obfuscation and ease of

implementation. Thus the algorithms must be computationally efficient, and not require substantial data structures.

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 traditionally reserved the following use of the 16-bit port range of TCP and UDP:

- o The Well Known Ports, 0 through 1023.
- o The Registered Ports, 1024 through 49151
- o 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.](#) 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 five-tuples is handled by some operating systems 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. Initialization value 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;
} while (count > 0);

return ERROR;
```

Figure 1

This algorithm works well provided that the number of connections 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 five-tuple collisions are rare.

However, this method has the drawback that the 'next_ephemeral' variable and thus the ephemeral port range is shared between all connections 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.

3. Randomizing the Ephemeral Ports

3.1. Characteristics of a good ephemeral port randomization algorithm

There are a number of factors to consider when designing a policy of selection of ephemeral ports, which include:

- o Minimizing the predictability of the ephemeral port numbers used for future connections.
- o Maximizing the port reuse cycle.
- o Avoiding conflict with applications that depend on the use of specific port numbers.

Given the goal of improving TCP's resistance to attack by obfuscation of the four-tuple that identifies a TCP connection, it is key to minimize the predictability of the ephemeral ports that will be selected for new connections. 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 a short reuse cycle of port numbers, which could lead to the interoperability problems discussed in . It is also worth noting that, provided adequate randomization algorithms are in use, the larger the range from which ephemeral ports are selected, the smaller the chances of an attacker are to guess the selected port number.

A number of implementations will not allow the creation of a new connection if there exists a previous incarnation of the same connection in any state other than the fictional state CLOSED. This can be problematic in scenarios in which a client establishes connections with a specific service at server at a high rate: even if the connections are also closed at a high rate, one of the systems (the one performing the active close) will keep each of the closed connection in the TIME-WAIT state for $2 \times \text{MSL}$. If the connection rate is high enough, at some point all the ephemeral ports at the client will be in use by some connection in the TIME-WAIT state, thus preventing the establishment of new connections. Therefore, the only strategy that can be relied upon to avoid this interoperability problem is to maximize the ephemeral port reuse cycle at a client, with the goal of reducing the chances that a previous incarnation of the same connection exists when a new connection is tried to be established. A good algorithm to maximize the port reuse cycle would consider the time a given ephemeral port number was last used, and would avoid reusing the last recently used port numbers. A simple approach to maximize the port reuse cycle 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 TCP connection. 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, transport protocols should avoid using those port numbers as ephemeral ports.

3.2. 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 an array of bits, in which each bit would correspond to each of the port numbers 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 therefore cannot be allocated. Thus, before allocating a port number, 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.

3.3. Ephemeral Port Randomization Algorithms

Transport protocols SHOULD allocate their ephemeral ports randomly, since this help 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 */
next_ephemeral = min_ephemeral + random()
                % (max_ephemeral - min_ephemeral + 1);

count = max_ephemeral - min_ephemeral + 1;

do {
    if(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'.

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 connections before a port can be chosen.

Although carefully chosen random sources and optimized five-tuple lookup mechanisms (e.g., optimized through hashing), will mitigate the cost of this verification, some systems may still not want to incur this search time.

Systems that may be specially susceptible to this kind of repeated five-tuple collisions are those that create many connections from a single local IP address to a single service (i.e. both of the IP addresses and the server port are fixed). Gateways such as proxy servers are an example of such a system.

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. 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, the connection request could possibly fail ([[Silbersack](#)] describes this problem for the TCP case).

Therefore, it is desirable to keep the port reuse frequency as low as possible.

3.3.2. Algorithm 2: Another simple port randomization algorithm

Another algorithm for selecting a random port number is shown in Figure 3, in which in the event a local connection-id collision is detected, another port number is selected randomly, as follows:

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

count = max_ephemeral - min_ephemeral + 1;

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

    next_ephemeral = min_ephemeral + random()
                    % (max_ephemeral - min_ephemeral + 1);
    count--;
} while (count > 0);

return ERROR;
```

Figure 3

We will refer to this algorithm as 'Algorithm 2'. The only difference between this algorithm and Algorithm 1 is that the search time for this variant may be longer than for the later, particularly when there are a large number of port numbers already in use.

3.3.3. Algorithm 3: Simple hash-based algorithm

We would like to achieve the port reuse properties of traditional BSD port selection algorithm, 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 ranges of the connections entirely. 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 code at system boot time. Initialization value could be r
next_ephemeral = 0;

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

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

    if(five-tuple is unique)
        return port;
} 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-connection 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 similar to that shown in Figure 3. 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 connection separation, however, they should also be included in the offset calculation.

Cryptographic algorithms stronger than e.g. MD5 should not be necessary, given that port randomization 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 and 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 include 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.

3.3.4. Algorithm 4: Double-hash randomization 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 */
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(five-tuple is unique)
        return port;
} while (count > 0);

return ERROR;
```

Figure 5

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

'table[]' could be initialized with random values, as indicated by the initialization code in Figure 5. G() would return a value between 0 and (TABLE_LENGTH-1) taking 'offset' as its input. G() could, for example, perform the exclusive-or (xor) operation between all the bytes in 'offset', or could be another cryptographic hash function such as that used in F().

The array 'table[]' assures that successive connections to the same 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 2. That is, a connection established 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 actual port sequence will result from adding the corresponding entry of 'table[]'

to the variable 'offset', which actually selects the actual port sequence (as in Algorithm 3).

3.4. Secret-key considerations for hash-based port randomization 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. Thus the change in secret key should be done with consideration and could be performed whenever one of the following events occur:

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

[3.5.](#) Choosing an ephemeral port randomization algorithm

The algorithm sketched in Figure 1 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.

Algorithm 1 and Algorithm 2 have the advantage that they provide complete randomization. However, they may increase the chances of port number collisions, which could lead to failure of the connection establishment attempts.

Algorithm 3 provides complete separation in local and remote IP addresses and remote port space, and only limited separation in other dimensions (See Section [Section 3.4](#)), and thus may scale better than Algorithm 1 and Algorithm 2. 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[]'`.

Finally, a special case that precludes 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 6

This code sequence results 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. This policy has been implemented by Linux [[Linux](#)].

4. Security Considerations

Randomizing 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 connection to be attacked could learn the IP addresses and port numbers in use (and also sequence numbers etc.) and easily attack the connection. Randomizing ports 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 connection), 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.

5. 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, Lars Eggert, Gorby Fairhurst, Alfred Hoenes, Carlos Pignataro, Joe Touch, 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.

[6.](#) References

[6.1.](#) 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.
- [RFC1948] Bellare, S., "Defending Against Sequence Number Attacks", [RFC 1948](#), May 1996.
- [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.
- [RFC2960] 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.
- [RFC3550] Schulzrinne, H., Casner, S., Frederick, R., and V. Jacobson, "RTP: A Transport Protocol for Real-Time Applications", STD 64, [RFC 3550](#), July 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.

6.2. 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-02](#) (work in progress),
May 2007.
- [RFC4953] Touch, J., "Defending TCP Against Spoofing Attacks",
[RFC 4953](#), July 2007.
- [Linux] The Linux Project, "<http://www.kernel.org>".
- [NetBSD] The NetBSD Project, "<http://www.netbsd.org>".
- [OpenBSD] The OpenBSD Project, "<http://www.openbsd.org>".
- [Silbersack]
Silbersack, M., "Improving TCP/IP security through
randomization without sacrificing interoperability.",
EuroBSDCon 2005 Conference , 2005.
- [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-00](#)
(work in progress), November 2007.
- [Watson] Watson, P., "Slipping in the Window: TCP Reset attacks",
december 2003.

[Appendix A](#). Survey of the algorithms in use by some popular implementations

[A.1](#). FreeBSD

FreeBSD implements Algorithm 2. with a 'min_port' of 49152 and a 'max_port' of 65535. If the selected port number is in use, the next available port number is tried next [[FreeBSD](#)].

[A.2](#). Linux

Linux implements Algorithm 3. If the algorithm is faced with the corner-case scenario described in [Section 3.5](#), Algorithm 2 is used instead [[Linux](#)].

[A.3](#). NetBSD

NetBSD does not randomize 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 implements Algorithm 2. with a 'min_port' of 1024 and a 'max_port' of 49151. If the selected port number is in use, the next available port number is tried next [[OpenBSD](#)].

[Appendix B](#). Changes from previous versions of the draft

[B.1](#). Changes from [draft-ietf-tsvwg-port-randomisation-00](#)

- o Added [Section 3.1](#).
- o Changed Intended Status from "Satandards Track" to "BCP".
- o Miscellaneous editorial changes.

[B.2](#). Changes from [draft-larsen-tsvwg-port-randomisation-02](#)

- o Draft resubmitted as [draft-ietf](#).
- o Included references and text on protocols other than TCP.
- o Added the second variant of the simple port randomization algorithm
- o Reorganized the algorithms into different sections
- o Miscellaneous editorial changes.

[B.3](#). Changes from [draft-larsen-tsvwg-port-randomisation-01](#)

- o No changes. Draft resubmitted after expiration.

[B.4](#). Changes from [draft-larsen-tsvwg-port-randomization-00](#)

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

[B.5](#). Changes from [draft-larsen-tsvwg-port-randomisation-00](#)

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

- o 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

Full Copyright Statement

Copyright (C) The IETF Trust (2008).

This document is subject to the rights, licenses and restrictions contained in [BCP 78](#), and except as set forth therein, the authors retain all their rights.

This document and the information contained herein are provided on an "AS IS" basis and THE CONTRIBUTOR, THE ORGANIZATION HE/SHE REPRESENTS OR IS SPONSORED BY (IF ANY), THE INTERNET SOCIETY, THE IETF TRUST AND THE INTERNET ENGINEERING TASK FORCE DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO ANY WARRANTY THAT THE USE OF THE INFORMATION HEREIN WILL NOT INFRINGE ANY RIGHTS OR ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE.

Intellectual Property

The IETF takes no position regarding the validity or scope of any Intellectual Property Rights or other rights that might be claimed to pertain to the implementation or use of the technology described in this document or the extent to which any license under such rights might or might not be available; nor does it represent that it has made any independent effort to identify any such rights. Information on the procedures with respect to rights in RFC documents can be found in [BCP 78](#) and [BCP 79](#).

Copies of IPR disclosures made to the IETF Secretariat and any assurances of licenses to be made available, or the result of an attempt made to obtain a general license or permission for the use of such proprietary rights by implementers or users of this specification can be obtained from the IETF on-line IPR repository at <http://www.ietf.org/ipr>.

The IETF invites any interested party to bring to its attention any copyrights, patents or patent applications, or other proprietary rights that may cover technology that may be required to implement this standard. Please address the information to the IETF at ietf-ipr@ietf.org.

Acknowledgment

Funding for the RFC Editor function is provided by the IETF Administrative Support Activity (IASA).