Internet Research Task Force (IRTF)
Internet-Draft
Intended status: Informational
Expires: July 10, 2021

F. Gont
SI6 Networks
I. Arce
Quarkslab

January 6, 2021

On the Generation of Transient Numeric Identifiers
draft-irtf-pearg-numeric-ids-generation-05

Abstract

This document performs an analysis of the security and privacy implications of different types of "transient numeric identifiers" used in IETF protocols, and tries to categorize them based on their interoperability requirements and the associated failure severity when such requirements are not met. Subsequently, it provides advice on possible algorithms that could be employed to satisfy the interoperability requirements of each identifier category, while minimizing the negative security and privacy implications, thus providing guidance to protocol designers and protocol implementers. Finally, it describes a number of algorithms that have been employed in real implementations to generate transient numeric identifiers and analyzes their security and privacy properties.

Status of This Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at https://datatracker.ietf.org/drafts/current/.

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."

This Internet-Draft will expire on July 10, 2021.

Table of Contents

1. Introduction

Network protocols employ a variety of transient numeric identifiers for different protocol entities, ranging from DNS Transaction IDs (TxIDs) to transport protocol ephemeral ports (e.g. TCP ephemeral ports) or IPv6 Interface Identifiers (IIDs). These identifiers usually have specific properties (e.g. uniqueness during a specified period of time) that must be satisfied such that they do not result in negative interoperability implications, and an associated failure severity when such properties are not met, ranging from soft to hard failures.

For more than 30 years, a large number of implementations of the TCP/IP protocol suite have been subject to a variety of attacks, with effects ranging from Denial of Service (DoS) or data injection, to information leakages that could be exploited for pervasive monitoring [RFC7258]. The root cause of these issues has been, in many cases, the poor selection of transient numeric identifiers in such protocols, usually as a result of insufficient or misleading specifications. While it is generally trivial to identify an algorithm that can satisfy the interoperability requirements of a given transient numeric identifier, empirical evidence exists that doing so without negatively affecting the security and/or privacy properties of the aforementioned protocols is prone to error [I-D.irtf-pearg-numeric-ids-history].

For example, implementations have been subject to security and/or privacy issues resulting from:

- Predictable TCP Initial Sequence Numbers (ISNs) [RFC0793]
- Predictable initial timestamp in TCP timestamps Options (TSval in SYN or SYN/ACK) [RFC7323]
- Predictable TCP ephemeral port numbers [RFC0793]
- Predictable IPv4 or IPv6 Fragment Identifiers (Fragment IDs) [RFC0791] [RFC8200]
- Predictable IPv6 Interface Identifiers (IIDs) [RFC4291]
- Predictable DNS Transaction Identifiers (TxIDs) [RFC1035]

Recent history indicates that when new protocols are standardized or new protocol implementations are produced, the security and privacy properties of the associated transient numeric identifiers tend to be overlooked, and inappropriate algorithms to generate transient numeric identifiers are either suggested in the specifications or selected by implementers. As a result, it should be evident that advice in this area is warranted.

This document contains a non-exhaustive survey of transient numeric identifiers employed in various IETF protocols, and aims to categorize such identifiers based on their interoperability requirements, and the associated failure severity when such requirements are not met. Subsequently, it provides advice on possible algorithms that could be employed to satisfy the interoperability requirements of each category, while minimizing negative security and privacy implications. Finally, it analyzes several algorithms that have been employed in real implementations to meet such requirements, and analyzes their security and privacy properties.

2. Terminology

Transient Numeric Identifier:
: A data object in a protocol specification that can be used to definitely distinguish a protocol object (a datagram, network interface, transport protocol endpoint, session, etc.) from all other objects of the same type, in a given context. Transient numeric identifiers are usually defined as a series of bits, and represented using integer values. These identifiers are typically dynamically selected, as opposed to statically-assigned numeric identifiers (see e.g. [IANA-PROT]). We note that different transient numeric identifiers may have additional requirements or properties depending on their specific use in a protocol. We use the term "transient numeric identifier" (or simply "numeric identifier" or "identifier" as short forms) as a generic term to refer to any data object in a protocol specification that satisfies the identification property stated above.

Failure Severity:
: The consequences of a failure to comply with the interoperability requirements of a given identifier. Severity considers the worst potential consequence of a failure, determined by the system damage and/or time lost to repair the failure. In this document we define two types of failure severity: "soft failure" and "hard failure".

Soft Failure:

A soft failure is a recoverable condition in which a protocol does not operate in the prescribed manner but normal operation can be resumed automatically in a short period of time. For example, a simple packet-loss event that is subsequently recovered with a packet-retransmission can be considered a soft failure.

Hard Failure:
A hard failure is a non-recoverable condition in which a protocol does not operate in the prescribed manner or it operates with excessive degradation of service. For example, an established TCP connection that is aborted due to an error condition constitutes, from the point of view of the transport protocol, a hard failure, since it enters a state from which normal operation cannot be resumed.

3. Threat Model

Throughout this document, we assume an attacker does not have physical or logical access to the device(s) being attacked, and does not have access to the packets being transferred between the sender and the receiver(s) of the target protocol (if any). However, we assume the attacker can send any traffic to the target device(s), to e.g. sample transient numeric identifiers employed by such device(s).

4. Issues with the Specification of Transient Numeric Identifiers

While assessing protocol specifications regarding the use of transient numeric identifiers, we have found that most of the issues discussed in this document arise as a result of one of the following conditions:

- Protocol specifications that under-specify the requirements for their transient numeric identifiers
- Protocol specifications that over-specify their transient numeric identifiers
- Protocol implementations that simply fail to comply with the specified requirements

A number of protocol specifications (too many of them) have simply overlooked the security and privacy implications of transient numeric identifiers [I-D.irtf-pearg-numeric-ids-history]. Examples of them are the specification of TCP ephemeral ports in [RFC0793], the specification of TCP sequence numbers in [RFC0793], or the specification of the DNS TxID in [RFC1035].

On the other hand, there are a number of protocol specifications that over-specify some of their associated transient numeric identifiers. For example, [RFC4291] essentially overloads the semantics of IPv6 Interface Identifiers (IIDs) by embedding link-layer addresses in the IPv6 IIDs, when the interoperability requirement of uniqueness could be achieved in other ways that do not result in negative security and privacy implications [RFC7721]. Similarly, [RFC2460] suggested the use of a global counter for the generation of Fragment Identification values, when the interoperability properties of uniqueness per {Src IP, Dst IP} could be achieved with other algorithms that do not result in negative security and privacy implications [RFC7739].

Finally, there are protocol implementations that simply fail to comply with existing protocol specifications. For example, some popular operating systems (notably Microsoft Windows) still fail to implement transport protocol ephemeral port randomization, as recommended in [RFC6056].

5. Protocol Failure Severity

Section 2 defines the concept of "Failure Severity", along with two types of failure severities that we employ throughout this document: soft and hard.

Our analysis of the severity of a failure is performed from the point of view of the protocol in question. However, the corresponding severity on the upper protocol (or application) might not be the same as that of the protocol in question. For example, a TCP connection that is aborted might or might not result in a hard failure of the upper application: if the upper application can establish a new TCP connection without any impact on the application, a hard failure at the TCP protocol may have no severity at the application level. On the other hand, if a hard failure of a TCP connection results in excessive degradation of service at the application layer, it will also result in a hard failure at the application.

6. Categorizing Transient Numeric Identifiers

This section includes a non-exhaustive survey of transient numeric identifiers, and proposes a number of categories that can accommodate these identifiers based on their interoperability requirements and their failure modes (soft or hard)

Identifier	Interoperability Requirements	Failure Severity
IPv6 Frag ID	Uniqueness (for IP address pair)	Soft/Hard (1)
IPv6 IID	Uniqueness (and stable within IPv6 prefix) (2)	Soft (3)
TCP ISN	Monotonically-increasing (4)	Hard (4)
TCP initial timestamps	Monotonically-increasing (5)	Hard (5)
TCP eph. port	Uniqueness (for connection ID)	Hard
IPv6 Flow Label	Uniqueness	None (6)
DNS TxID	Uniqueness	None (7)

Table 1: Survey of Transient Numeric Identifiers

Notes:

(1)
While a single collision of Fragment ID values would simply lead to a single packet drop (and hence a "soft" failure), repeated collisions at high data rates might trash the Fragment ID space, leading to a hard failure [RFC4963].

(2)
While the interoperability requirements are simply that the Interface ID results in a unique IPv6 address, for operational reasons it is typically desirable that the resulting IPv6 address (and hence the corresponding Interface ID) be stable within each network [RFC7217] [RFC8064].

(3)
While IPv6 Interface IDs must result in unique IPv6 addresses, IPv6 Duplicate Address Detection (DAD) [RFC4862] allows for the detection of duplicate addresses, and hence such Interface ID collisions can be recovered.

(4)
In theory, there are no interoperability requirements for TCP Initial Sequence Numbers (ISNs), since the TIME-WAIT state and

TCP's "quiet time" concept take care of old segments from previous incarnations of a connection. However, a widespread optimization allows for a new incarnation of a previous connection to be created if the ISN of the incoming SYN is larger than the last sequence number seen in that direction for the previous incarnation of the connection. Thus, monotonically-increasing TCP sequence numbers allow for such optimization to work as expected [RFC6528], and can help avoid connection-establishment failures.

(5)
Strictly speaking, there are no interoperability requirements for the *initial* TCP timestamp employed by a TCP (i.e., the TS Value (TSval) in a segment with the SYN bit set). However, a some TCP implementations allow a new incarnation of a previous connection to be created if the TSval of the incoming SYN is larger than the last TSval seen in that direction for the previous incarnation of the connection (please see [RFC6191]). Thus, monotonically-increasing TCP initial timestamps (across connections to the same endpoint) allow for such optimization to work as expected [RFC6191], and can help avoid connection-establishment failures.

(6)
The IPv6 Flow Label is typically employed for load sharing [RFC7098], along with the Source and Destination IPv6 addresses. Reuse of a Flow Label value for the same set {Source Address, Destination Address} would typically cause both flows to be multiplexed onto the same link. However, as long as this does not occur deterministically, it will not result in any negative implications.

(7)
DNS TxIDs are employed, together with the Source Address, Destination Address, Source Port, and Destination Port, to match DNS requests and responses. However, since an implementation knows which DNS requests were sent for that set of {Source Address, Destination Address, Source Port, and Destination Port, DNS TxID}, a collision of TxID would result, if anything, in a small performance penalty (the response would nevertheless be discarded when it is found that it does not answer the query sent in the corresponding DNS query).

Based on the survey above, we can categorize identifiers as follows:

Cat #	Category	Sample Proto IDs
1	Uniqueness (soft failure)	IPv6 Flow L., DNS TxIDs
2	Uniqueness (hard failure)	IPv6 Frag ID, TCP ephemeral port
3	Uniqueness, stable within context (soft failure)	IPv6 IIDs
4	Uniqueness, monotonically increasing within context (hard failure)	TCP ISN, TCP initial timestamps

Table 2: Identifier Categories

We note that Category #4 could be considered a generalized case of category #3, in which a monotonically increasing element is added to a stable (within context) element, such that the resulting identifiers are monotonically increasing within a specified context. That is, the same algorithm could be employed for both #3 and #4, given appropriate parameters.

7. Common Algorithms for Transient Numeric Identifier Generation

The following subsections describe some sample algorithms that can be employed for generating transient numeric identifiers for each of the categories above.

All of the variables employed in the algorithms of the following subsections are of "unsigned integer" type, except for the "retry" variable, that is of (signed) "integer" type.

7.1. Category #1: Uniqueness (soft failure)

The requirement of uniqueness with a soft failure mode can be complied with a Pseudo-Random Number Generator (PRNG).

We note that since the premise is that collisions of numeric identifiers of this category only leads to soft failures, in many (if not most) cases, the algorithm will not need to check the suitability of a selected identifier (i.e., in such cases check_suitable_id() could always return "true").

In scenarios where e.g. simultaneous use of a given numeric ID is undesirable and the implementation detects such condition, an implementation may opt to select the next available identifier in the same sequence, or select another random number. Section 7.1.1 is an implementation of the former strategy, while Section 7.1.2 is an implementation of the later.

7.1.1. Simple Randomization Algorithm

```
/* Transient Numeric ID selection function */

id_range = max_id - min_id + 1;
next_id = min_id + (random() % id_range);
retry = id_range;

do {
    if (check_suitable_id(next_id)) {
        return next_id;
    }

    if (next_id == max_id) {
        next_id = min_id;
    } else {
        next_id++;
    }

    retry--;

} while (retry > 0);

return ERROR;
```

NOTES:

random() is a function that returns a pseudo-random unsigned integer number of appropriate size. Note that the output needs to be unpredictable, and typical implementations of the POSIX random() function do not necessarily meet this requirement. See [RFC4086] for randomness requirements for security. Beware that "adapting" the length of the output of random() with a modulo operator (e.g., C language's "%") may change the distribution of the PRNG.

The function check_suitable_id() can check, when possible and desirable, whether this identifier is suitable (e.g. it is not already in use). Depending on how/where the numeric identifier is used, it may or may not be possible (or even desirable) to check whether the numeric identifier is in use (or whether it has been recently employed). When an identifier is found to be unsuitable,

this algorithm selects the next available numeric identifier in sequence.

Even when this algorithm selects numeric IDs randomly, it is biased towards the first available numeric ID after a sequence of unavailable numeric IDs. For example, if this algorithm is employed for transport protocol ephemeral port randomization [RFC6056] and the local list of unsuitable port numbers (e.g., registered port numbers that should not be used for ephemeral ports) is significant, an attacker may actually have a significantly better chance of guessing a port number.

All the variables (in this and all the algorithms discussed in this document) are unsigned integers.

Assuming the randomness requirements for the PRNG are met (see [RFC4086]), this algorithm does not suffer from any of the issues discussed in Section 8.

7.1.2. Another Simple Randomization Algorithm

The following pseudo-code illustrates another algorithm for selecting a random numeric identifier which, in the event a selected identifier is found to be unsuitable (e.g., already in use), another identifier is randomly selected:

```
/* Transient Numeric ID selection function */

id_range = max_id - min_id + 1;
retry = id_range;

do {
    next_id = min_id + (random() % id_range);

    if (check_suitable_id(next_id)) {
        return next_id;
    }

    retry--;

} while (retry > 0);

return ERROR;
```

This algorithm might be unable to select an identifier (i.e., return "ERROR") even if there are suitable identifiers available, in cases

where a large number of identifiers are found to be unsuitable (e.g. "in use").

The same considerations from Section 7.1.1 with respect to the properties of random() and the adaptation of its output length apply to this algorithm.

Assuming the randomness requirements for the PRNG are met (see [RFC4086]), this algorithm does not suffer from any of the issues discussed in Section 8.

7.2. Category #2: Uniqueness (hard failure)

One of the most trivial approaches for achieving uniqueness for an identifier (with a hard failure mode) is to reduce the identifier reuse frequency by generating the numeric identifiers with a monotonically-increasing function (e.g. linear). As a result, any of the algorithms described in Section 7.4 ("Category #4: Uniqueness, monotonically increasing within context (hard failure)") can be readily employed for complying with the requirements of this numeric identifier category.

In cases where suitability (e.g. uniqueness) of the selected identifiers can be definitely assessed by the local system, any of the algorithms described in Section 7.1 ("Category #1: Uniqueness (soft failure)") can be readily employed for complying with the requirements of this numeric identifier category.

NOTE:

In the case of e.g. TCP ephemeral ports or TCP ISNs, a transient numeric identifier that might seem suitable from the perspective of the local system, might actually be unsuitable from the perspective of the remote system (e.g., because there is state associated with the selected identifier at the remote system). Therefore, in these cases it is not possible employ the algorithms from Section 7.1 ("Category #1: Uniqueness (soft failure)").

7.3. Category #3: Uniqueness, stable within context (soft failure)

The goal of the following algorithm is to produce identifiers that are stable for a given context (identified by "CONTEXT"), but that change when the aforementioned context changes.

In order to avoid storing in memory the numeric identifier computed for each CONTEXT value, the following algorithm employs a calculated technique (as opposed to keeping state in memory) to generate a stable identifier for each given context.

```
    /* Transient Numeric ID selection function  */

    id_range = max_id - min_id + 1;

    retry = 0;

    do {
        offset = F(CONTEXT, retry, secret_key);
        next_id = min_id + (offset % id_range);

        if (check_suitable_id(next_id)) {
            return next_id;
        }

        retry++;

    } while (retry <= MAX_RETRIES);

    return ERROR;
```

In the following algorithm, the function F() provides a stateless and stable per-CONTEXT numeric identifier, where CONTEXT is the concatenation of all the elements that define the given context.

For example, if this algorithm is expected to produce IPv6 IIDs that are unique per network interface and SLAAC autoconfiguration prefix, the CONTEXT should be the concatenation of e.g. the network interface index and the SLAAC autoconfiguration prefix (please see [RFC7217] for an implementation of this algorithm for generation of stable IPv6 IIDs).

F() must be a cryptographically-secure hash function (e.g. SHA-256 [FIPS-SHS]), that is computed over the concatenation of its arguments. The result of F() is no more secure than the secret key, and therefore 'secret_key' must be unknown to the attacker, and must be of a reasonable length. 'secret_key' must remain stable for a given CONTEXT, since otherwise the numeric identifiers generated by this algorithm would not have the desired stability properties (i.e., stable for a given CONTEXT). In most cases, 'secret_key' can be selected with a PRNG (see [RFC4086] for recommendations on choosing secrets) at an appropriate time, and stored in stable or volatile storage for future use.

The result of F() is stored in the variable 'offset', which may take any value within the storage type range, since we are restricting the resulting identifier to be in the range [min_id, max_id] in a similar way as in the algorithm described in Section 7.1.1.

check_suitable_id() checks that the candidate identifier has suitable uniqueness properties. Collisions (i.e., an identifier that is not unique) are recovered by incrementing the 'retry' variable and recomputing F(), up to a maximum of MAX_RETRIES times. However, recovering from collisions will usually result in identifiers that fail to remain constant for the specified context. This is normally acceptable when the probability of collisions is small, as in the case of e.g. IPv6 IIDs resulting from SLAAC [RFC7217] [RFC4941].

For obvious reasons, the transient numeric identifiers generated with this algorithm allow for network activity correlation within "CONTEXT". However, this is essentially a design goal of this category of transient numeric identifiers.

7.4. Category #4: Uniqueness, monotonically increasing within context (hard failure)

7.4.1. Per-context Counter Algorithm

One possible way to select unique monotonically-increasing identifiers is to employ a per-context counter. Such an algorithm could be described as follows:

```
      /* Transient Numeric ID selection function */

      id_range = max_id - min_id + 1;
      retry = id_range;
      id_inc = increment() % id_range;

      if( (next_id = lookup_counter(CONTEXT)) == ERROR){
           next_id = min_id + random() % id_range;
      }

      do {
          if ( (max_id - next_id) >= id_inc){
              next_id = next_id + id_inc;
          }
          else {
              next_id = min_id + id_inc - (max_id - next_id);
          }

          if (check_suitable_id(next_id)){
              store_counter(CONTEXT, next_id);
              return next_id;
          }

          retry = retry - id_inc;

      } while (retry > 0);

      return ERROR;
```

NOTES:

increment() returns a small integer that is employed to increment the current counter value to obtain a numeric ID. This value must be much smaller than the number of possible values for the numeric IDs (i.e., "id_range"). Most implementations of this algorithm employ a constant increment of 1. Using a value other than 1 may help mitigate some information leakages (please see below), at the expense of a possible increase in the numeric ID reuse frequency.

The code above makes sure that the increment employed in the algorithm (id_inc) is always smaller than the number of possible values for the numeric IDs (i.e., "max_id - min_d + 1"). However, as noted above, this value must also be much smaller than the number of possible values for the numeric IDs.

lookup_counter() returns the current counter for a given context, or an error condition if that counter does not exist.

store_counter() saves a counter value for a given context.

check_suitable_id() is a function that checks whether the resulting identifier is acceptable (e.g., whether it is not already in use, etc.).

Essentially, whenever a new identifier is to be selected, the algorithm checks whether there there is a counter for the corresponding context. If there is, the value of such counter is incremented to obtain the new identifier, and the counter is updated. If no counter exists for such context, a new counter is created and initialized to a random value, and used as the new identifier. This algorithm produces a per-context counter, which results in one monotonically-increasing function for each context. Since each counter is initialized to a random value, the resulting values are unpredictable by an off-path attacker.

The choice of id_inc has implications on both the security and privacy properties of the resulting identifiers, but also on the corresponding interoperability properties. On one hand, minimizing the increments generally minimizes the identifier reuse frequency, albeit at increased predictability. On the other hand, if the increments are randomized, predictability of the resulting identifiers is reduced, and the information leakage produced by global constant increments is mitigated. However, using larger increments than necessary can result in higher identifier reuse frequency.

This algorithm has the following drawbacks:

- This algorithm requires an implementation to store each per-CONTEXT counter in memory. If, as a result of resource management, the counter for a given context must be removed, the last identifier value used for that context will be lost. Thus, if subsequently an identifier needs to be generated for the same context, that counter will need to be recreated and reinitialized to random value, thus possibly leading to reuse/collision of numeric identifiers.

- Keeping one counter for each possible "context" may in some cases be considered too onerous in terms of memory requirements.

Otherwise, the identifiers produced by this algorithm do not suffer from the other issues discussed in Section 8.

7.4.2. Simple Hash-Based Algorithm

The goal of this algorithm is to produce monotonically-increasing sequences, with a randomized initial value, for each given context. For example, if the identifiers being generated must be

monotonically-increasing for each {src IP, dst IP} set, then each possible combination of {src IP, dst IP} should have a separate monotonically-increasing sequence, that starts at a different random value.

Instead of maintaining a per-context counter (as in the algorithm from Section 7.4.1), the following algorithm employs a calculated technique to maintain a random offset for each possible context.

In the following algorithm, the function F() provides a (stateless) unpredictable offset for each given context (as identified by 'CONTEXT').

```
    /* Initialization code */
    counter = 0;

    /* Transient Numeric ID selection function  */

    id_range = max_id - min_id + 1;
    id_inc = increment() % id_range;
    offset = F(CONTEXT, secret_key);
    retry = id_range;

    do {
        next_id = min_id + (offset + counter) % id_range;
        counter = counter + id_inc;

        if (check_suitable_id(next_id)) {
            return next_id;
        }

        retry = retry - id_inc;

    } while (retry > 0);

    return ERROR;
```

The function F() provides a "per-CONTEXT" fixed offset within the numeric identifier "space". Both the 'offset' and 'counter' variables may take any value within the storage type range since we are restricting the resulting identifier to be in the range [min_id, max_id] in a similar way as in the algorithm described in Section 7.1.1. This allows us to simply increment the 'counter' variable and rely on the unsigned integer to wrap around.

The function F() should be a cryptographically-secure hash function (e.g. SHA-256 [FIPS-SHS]). CONTEXT is the concatenation of all the

elements that define a given context. For example, if this algorithm is expected to produce identifiers that are monotonically-increasing for each set (Source IP Address, Destination IP Address), CONTEXT should be the concatenation of these two IP addresses.

The result of F() is no more secure than the secret key, and therefore 'secret_key' must be unknown to the attacker, and must be of a reasonable length. 'secret_key' must remain stable for a given CONTEXT, since otherwise the numeric identifiers generated by this algorithm would not have the desired stability properties (i.e., monotonically-increasing for a given CONTEXT). In most cases, 'secret_key' can be selected with a PRNG (see [RFC4086] for recommendations on choosing secrets) at an appropriate time, and stored in stable or volatile storage for future use.

It should be noted that, since this algorithm uses a global counter ("counter") for selecting identifiers (i.e., all counters share the same increments space), this algorithm produces an information leakage (as described in Section 8.2). For example, if this algorithm were used for selecting TCP ephemeral ports, and an attacker could force a client to periodically establish 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 five-tuple collisions, of course). This information leakage could be partially mitigated by employing small random values for the increments (i.e., increment() function), instead of the constant "1".

We nevertheless note that an improved mitigation of this information leakage would result from employing the algorithm from Section 7.4.3, instead.

7.4.3. Double-Hash Algorithm

A trade-off between maintaining a single global 'counter' variable and maintaining 2**N 'counter' 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 integers, which would provide a separation of the increment space into multiple buckets. This improvement could be incorporated into the algorithm from Section 7.4.2 as follows:

```
    /* Initialization code */

    for(i = 0; i < TABLE_LENGTH; i++) {
        table[i] = random();
    }

    /* Transient Numeric ID selection function */

    id_range = max_id - min_id + 1;
    id_inc = increment() % id_range;
    offset = F(CONTEXT, secret_key1);
    index = G(CONTEXT, secret_key2) % TABLE_LENGTH;
    retry = id_range;

    do {
        next_id = min_id + (offset + table[index]) % id_range;
        table[index] = table[index] + id_inc;

        if (check_suitable_id(next_id)) {
            return next_id;
        }

       retry = retry - id_inc;

    } while (retry > 0);

    return ERROR;
```

'table[]' could be initialized with random values, as indicated by the initialization code in the pseudo-code above.

Both F() and G() should be cryptographically-secure hash functions (e.g. SHA-256 [FIPS-SHS]) computed over the concatenation of each of their respective arguments. Both F() and G() would employ the same CONTEXT (the concatenation of all the elements that define a given context), and would use separate secret keys (secret_key1, and secret_key2, respectively).

The results of F() and G() are no more secure than their respective secret keys ('secret_key1' and 'secret_key2', respectively), and therefore both secret keys must be unknown to the attacker, and must be of a reasonable length. Both secret keys must remain stable for the given CONTEXT, since otherwise the numeric identifiers generated by this algorithm would not have the desired stability properties (i.e., monotonically-increasing for a given CONTEXT). In most cases, both secret keys can be selected with a PRNG (see [RFC4086] for

recommendations on choosing secrets) at an appropriate time, and stored in stable or volatile storage for future use.

The array 'table[]' assures that successive identifiers for a given context will be monotonically-increasing. Since the increments space is separated into TABLE_LENGTH different spaces, the identifier reuse frequency will be (probabilistically) lower than that of the algorithm in Section 7.4.2. That is, the generation of an identifier for one given context will not necessarily result in increments in the identifier sequence for other contexts. It is interesting to note that the size of 'table[]' does not limit the number of different identifier sequences, but rather separates the *increments* into TABLE_LENGTH different spaces. The identifier sequence will result from adding the corresponding entry from 'table[]' to the variable 'offset', which selects the actual identifier sequence (as in the algorithm from Section 7.4.2).

An attacker can perform traffic analysis for any "increment space" (i.e., context) into which the attacker has "visibility" -- namely, the attacker can force a node to generate identifiers where G(CONTEXT, secret_key2) identifies the target "increment space". However, the attacker's ability to perform traffic analysis is very reduced when compared to the predictable linear identifiers (described in Appendix A.1) and the hash-based identifiers (described in Section 7.4.2). 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) and/or by randomizing the increments (i.e., increment() returning a small random number as opposed to the constant 1).

Otherwise, this algorithm does not suffer from the issues discussed in Section 8.

8. Common Vulnerabilities Associated with Transient Numeric Identifiers

8.1. Network Activity Correlation

An identifier that is predictable within a given context allows for network activity correlation within that context.

For example, a stable IPv6 Interface Identifier allows for network activity to be correlated for the context in which that Interface Identifier is stable [RFC7721]. A stable-per-network IPv6 Interface Identifier (as in [RFC7217]) allows for network activity correlation within a network, whereas a constant IPv6 Interface Identifier (that remains constant across networks) allows not only network activity correlation within the same network, but also across networks ("host tracking").

Similarly, a node that generates TCP ISNs with a global counter could allow network activity correlation across networks, since the communicating nodes could infer the identity of the node based on the TCP ISNs employed for subsequent communication instances. Similarly, a node that generates predictable IPv6 Fragment Identification values could be subject to network activity correlation (see e.g. [Bellovin2002]).

8.2. Information Leakage

Transient numeric identifiers that are not randomized can leak out information to other communicating nodes. For example, it is common to generate identifiers like:

```
ID = offset(CONTEXT) + mono(CONTEXT);
```

This generic expression generates identifiers by adding a monotonically-increasing function (e.g. linear) to a randomized offset. offset() is constant within a given context, whereas mono() is monotonically-increasing function for a given context. Identifiers generated with this expression will generally be predictable within CONTEXT.

On the other hand, mono() will result in a monotonically-increasing sequence within CONTEXT. The predictability of mono(), irrespective of the predictability of offset(), can leak out information can be of use to attackers. For example, a node that selects ephemeral port numbers as in:

```
ephemeral_port = offset(Dest_IP) + mono()
```

that is, with a per-destination offset, but global mono() function (e.g., a global counter), will leak information about the number of outgoing connections that have been issued between any two issued outgoing connections.

Similarly, a node that generates Fragment Identification values as in:

```
Frag_ID = offset(Src_IP, Dst_IP) + mono()
```

will leak out information about the number of fragmented packets that have been transmitted between any two other transmitted fragmented packets. The vulnerabilities described in [Sanfilippo1998a], [Sanfilippo1998b], and [Sanfilippo1999] are all associated with the use of a global mono() function (i.e., irrespective of CONTEXT) -- particularly when it is a linear function (constant increments of 1 for each selected transient numeric identifier).

Predicting transient numeric identifiers can be of help for other types of attacks. For example, predictable TCP ISNs open the door to trivial connection-reset and data injection attacks (see e.g. [Joncheray1995].

8.3. Fingerprinting

Fingerprinting is the capability of an attacker to identify or re-identify a visiting user, user agent or device via configuration settings or other observable characteristics. Observable protocol objects and characteristics can be employed to identify/re-identify a variety of entities, ranging from the underlying hardware or Operating System (vendor, type and version), to the user itself (i.e. his/her identity). [EFF] illustrates web browser-based fingerprinting, but similar techniques can be applied at other layers and protocols, whether alternatively or in conjunction with it.

Transient numeric identifiers are one of the observable protocol components that could be leveraged for fingerprinting purposes. That is, an attacker could sample transient numeric identifiers to infer the algorithm (and its associated parameters, if any) for generating such identifiers, possibly revealing the underlying Operating System (OS) vendor, type, and version. This information could possibly be further leveraged in conjunction with other fingerprinting techniques and sources.

Evasion of protocol-stack fingerprinting can prove to be a very difficult task: most systems make use of a variety of protocols, each of which have a large number of parameters that can be set to arbitrary values or generated with a variety of algorithms with multiple parameters.

NOTE:

General protocol-based fingerprinting is discussed in [RFC6973], along with guidelines to mitigate the associated vulnerability. [Fyodor1998] and [Fyodor2006] are classic references on Operating System detection via protocol-based fingerprinting. Nmap [nmap] is probably the most popular tool for remote OS detection via active TCP/IP stack fingerprinting. p0f [Zalewski2012], on the other hand, is a tool for performing remote OS detection via passive TCP/IP stack fingerprinting. Finally, [TBIT] is a TCP fingerprinting tool that aims at characterizing the behavior of a remote TCP peer based on active probes, and which has been widely used in the research community.

Algorithms that, from the perspective of an observer (e.g., the legitimate communicating peer), result in specific values or patterns, will allow for at least some level of fingerprinting. For

example, the algorithm from Section 7.3 will typically allow fingerprinting within the context where the resulting identifiers are stable. Similarly, the algorithms from Section 7.4 will result in a monotonically-increasing sequences within a given context, thus allowing for at least some level of fingerprinting (when the other communicating entity can correlate different sampled identifiers as belonging to the same monotonically-increasing sequence).

Thus, where possible, algorithms from Section 7.1 should be preferred over algorithms that result in specific values or patterns.

8.4. Exploitation of the Semantics of Transient Numeric Identifiers

Identifiers that are not semantically opaque tend to be more predictable than semantically-opaque identifiers. For example, a MAC address contains an OUI (Organizationally-Unique Identifier) which identifies the vendor that manufactured the network interface card. This fact may be leveraged by an attacker trying to "guess" MAC addresses, who has some knowledge about the possible NIC vendor.

[RFC7707] discusses a number of techniques to reduce the search space when performing IPv6 address-scanning attacks by leveraging the semantics of the IIDs produced by traditional IID-generation algorithms that embed MAC addresses (now replaced by [RFC7217]).

8.5. Exploitation of Collisions of Transient Numeric Identifiers

In many cases, the collision of transient network identifiers can have a hard failure severity (or result in a hard failure severity if an attacker can cause multiple collisions deterministically, one after another). For example, predictable Fragment Identification values open the door to Denial of Service (DoS) attacks (see e.g. [RFC5722].).

8.6. Cryptanalysis

A number of algorithms discussed in this document (such as Section 7.4.2 and Section 7.4.3) rely on cryptographically-secure hash functions. Implementations that employ weak hash functions and keys of inappropriate size may be subject to cryptanalysis, where an attacker may be able to obtain the secret key employed for the hash algorithms, predict numeric identifiers, etc.

Furthermore, an implementation that overloads the semantics of the secret key may result in more trivial cryptanalysis, possibly resulting in the leakage of the value employed for the secret key.

NOTE:

[IPID-DEV] describes two vulnerable numeric ID generators that employ cryptographically-weak hash functions. Additionally, one of such implementations employs 32-bits of a kernel address as the secret key for a hash function, and therefore successful cryptanalysis leaks the aforementioned kernel address, allowing for Kernel Address Space Layout Randomization (KASLR) [KASLR] bypass.

9. Vulnerability Analysis of Specific Transient Numeric Identifiers Categories

The following subsections analyze common vulnerabilities associated with the generation of identifiers for each of the categories identified in Section 6.

9.1. Category #1: Uniqueness (soft failure)

Possible vulnerabilities associated with the algorithms from Section 7.1 include:

- Use of flawed PRNGs (please see e.g. [Zalewski2001], [Zalewski2002] and [Klein2007])

An implementer should consult [RFC4086] regarding randomness requirements for security, and consult relevant documentation when employing a PRNG provided by the underlying system.

Use of algorithms other than PRNGs for generating identifiers of this category is discouraged.

9.2. Category #2: Uniqueness (hard failure)

As noted in Section 7.2, this category can employ the same algorithms as Category #4, since a monotonically-increasing sequence tends to minimize the identifier reuse frequency. Therefore, the vulnerability analysis of Section 9.4 applies to this category.

Additionally, as noted in Section 7.2, some identifiers of this category might be able to use the algorithms from Section 7.1, in which case the same considerations from Section 9.1 apply.

9.3. Category #3: Uniqueness, stable within context (soft failure)

Possible vulnerabilities associated with the algorithms from Section 7.3 are:

1. Use cryptographically-weak hash functions, or inappropriate secret keys (whether inappropriate selection or inappropriate

size) that might allow for cryptanalysis, which could eventually be exploited by an attacker to predict future numeric identifiers.

2. Fingerprinting within the specified context.

9.4. Category #4: Uniqueness, monotonically increasing within context (hard failure)

A simple way to generalize algorithms employed for generating identifiers of Category #4 would be as follows:

```
/* Transient Numeric ID selection function */

id_range = max_id - min_id + 1;
retry = id_range;
id_inc = increment() % id_range;

do {
    update_mono(CONTEXT, id_inc);
    next_id = min_id + (offset(CONTEXT) + \
                        mono(CONTEXT)) % id_range;

    if (check_suitable_id(next_id)) {
        return next_id;
    }

    retry = retry - id_inc;

} while (retry > 0);

return ERROR;
```

NOTES:

increment() returns a small integer that is employed to generate a monotonically-increasing function. Some implementations employ a constant value for "increment()" (usually 1). The value returned by increment() must be much smaller than the value computed for "id_range".

update_mono(CONTEXT, id_inc) increments the counter corresponding to CONTEXT by "id_inc".

mono(CONTEXT) reads the counter corresponding to CONTEXT.

Essentially, an identifier (next_id) is generated by adding a monotonically-increasing function (mono()) to an offset value, unknown to the attacker and stable for given context (CONTEXT).

The following aspects of the algorithm should be considered:

- For the most part, it is the offset() function that results in identifiers that are unpredictable by an off-patch attacker. While the resulting sequence is known to be monotonically-increasing, the use of an randomized offset value makes the resulting values unknown to the attacker.

- The most straightforward "stateless" implementation of offset() is with a cryptographically-secure hash function that takes the values that identify the context and a "secret_key" (not shown in the figure above) as arguments.

- One possible implementation approach for mono() is to maintain per-context counters, initialized to random values. When a new identifier is to be selected, the corresponding counter is looked-up (based on the context) and incremented, to obtain a new transient numeric identifier. For example, the algorithm in Section 7.4.1 could be such an implementation of mono(). Another possible implementation of mono() would be to have mono() internally employ a single counter (as in the algorithm from Section 7.4.2), or map the increments for different contexts into a number of counters/buckets, such that the number of counters that need to be maintained in memory is reduced (as in the algorithm from Section 7.4.3).

- In all cases, a monotonically-increasing function is implemented by incrementing the previous value of the counter by increment(). In the most trivial case, increment() could return the constant "1". But increment() could also be implemented to return small random integers such that the increments are unpredictable (see Appendix A of [RFC7739]). This represents a trade-off between the unpredictability of the resulting transient numeric IDs and the transient numeric ID reuse frequency.

Considering the generic algorithm illustrated above, we can identify the following possible vulnerabilities:

- Since the algorithms for this category are similar to those of Section 9.3, with the addition of a monotonically-increasing function, all the issues discussed in Section 9.3 ("Category #3: Uniqueness, stable within context (soft failure)") also apply to this case.

- mono() can be correlated to the number of identifiers generated for a given context (CONTEXT). Thus, if mono() spans more than the necessary context, the "increments" could be leaked to other parties, thus disclosing information about the number of identifiers generated for CONTEXT. This is particularly the case when the algorithm employs a constant increment of 1. For example, an implementation where mono() is actually a single global counter, will unnecessarily leak information about the number of transient numeric identifiers that have been generated. [Fyodor2004] is one example of how such information leakages can be exploited. However, limiting the span of the increments space will require a larger number of counters to be stored in memory (i.e., a larger size for the TABLE_LENGTH parameter of the algorithm in Section 7.4.3.

- Transient numeric identifiers generated with this type of algorithm will normally allow for fingerprinting within CONTEXT since, for such context, the resulting identifiers will have an identifiable pattern (i.e. a monotonically-increasing sequence).

10. 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.

11. Security Considerations

The entire document is about the security and privacy implications of transient numeric identifiers. [I-D.gont-numeric-ids-sec-considerations] recommends that protocol specifications specify the interoperability requirements of their transient numeric identifiers, and include an assessment of the security and privacy implications of their transient numeric identifiers. This document analyzes possible algorithms (and their implications) that could be employed to comply with the interoperability properties of transient numeric identifiers, while minimizing the associated negative security and privacy implications.

12. Acknowledgements

The authors would like to thank (in alphabetical order) Bernard Aboba, Steven Bellovin, Luis Leon Cardenas Graide, Guillermo Gont, Joseph Lorenzo Hall, Gre Norcie, Shivan Sahib, and Martin Thomson, and Michael Tuexen, for providing valuable comments on earlier versions of this document.

The authors would like to thank Diego Armando Maradona for his magic and inspiration.

13. References

13.1. Normative References

[RFC0791] Postel, J., "Internet Protocol", STD 5, RFC 791, DOI 10.17487/RFC0791, September 1981, <https://www.rfc-editor.org/info/rfc791>.

[RFC0793] Postel, J., "Transmission Control Protocol", STD 7, RFC 793, DOI 10.17487/RFC0793, September 1981, <https://www.rfc-editor.org/info/rfc793>.

[RFC1035] Mockapetris, P., "Domain names - implementation and specification", STD 13, RFC 1035, DOI 10.17487/RFC1035, November 1987, <https://www.rfc-editor.org/info/rfc1035>.

[RFC2460] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", RFC 2460, DOI 10.17487/RFC2460, December 1998, <https://www.rfc-editor.org/info/rfc2460>.

[RFC4086] Eastlake 3rd, D., Schiller, J., and S. Crocker, "Randomness Requirements for Security", BCP 106, RFC 4086, DOI 10.17487/RFC4086, June 2005, <https://www.rfc-editor.org/info/rfc4086>.

[RFC4291] Hinden, R. and S. Deering, "IP Version 6 Addressing Architecture", RFC 4291, DOI 10.17487/RFC4291, February 2006, <https://www.rfc-editor.org/info/rfc4291>.

[RFC4862] Thomson, S., Narten, T., and T. Jinmei, "IPv6 Stateless Address Autoconfiguration", RFC 4862, DOI 10.17487/RFC4862, September 2007, <https://www.rfc-editor.org/info/rfc4862>.

[RFC4941] Narten, T., Draves, R., and S. Krishnan, "Privacy Extensions for Stateless Address Autoconfiguration in IPv6", RFC 4941, DOI 10.17487/RFC4941, September 2007, <https://www.rfc-editor.org/info/rfc4941>.

[RFC5722] Krishnan, S., "Handling of Overlapping IPv6 Fragments", RFC 5722, DOI 10.17487/RFC5722, December 2009, <https://www.rfc-editor.org/info/rfc5722>.

[RFC6056] Larsen, M. and F. Gont, "Recommendations for Transport-Protocol Port Randomization", BCP 156, RFC 6056, DOI 10.17487/RFC6056, January 2011, <https://www.rfc-editor.org/info/rfc6056>.

[RFC6191] Gont, F., "Reducing the TIME-WAIT State Using TCP Timestamps", BCP 159, RFC 6191, DOI 10.17487/RFC6191, April 2011, <https://www.rfc-editor.org/info/rfc6191>.

[RFC6528] Gont, F. and S. Bellovin, "Defending against Sequence Number Attacks", RFC 6528, DOI 10.17487/RFC6528, February 2012, <https://www.rfc-editor.org/info/rfc6528>.

[RFC7217] Gont, F., "A Method for Generating Semantically Opaque Interface Identifiers with IPv6 Stateless Address Autoconfiguration (SLAAC)", RFC 7217, DOI 10.17487/RFC7217, April 2014, <https://www.rfc-editor.org/info/rfc7217>.

[RFC7323] Borman, D., Braden, B., Jacobson, V., and R. Scheffenegger, Ed., "TCP Extensions for High Performance", RFC 7323, DOI 10.17487/RFC7323, September 2014, <https://www.rfc-editor.org/info/rfc7323>.

[RFC8064] Gont, F., Cooper, A., Thaler, D., and W. Liu, "Recommendation on Stable IPv6 Interface Identifiers", RFC 8064, DOI 10.17487/RFC8064, February 2017, <https://www.rfc-editor.org/info/rfc8064>.

[RFC8200] Deering, S. and R. Hinden, "Internet Protocol, Version 6 (IPv6) Specification", STD 86, RFC 8200, DOI 10.17487/RFC8200, July 2017, <https://www.rfc-editor.org/info/rfc8200>.

13.2. Informative References

[Bellovin2002]
Bellovin, S., "A Technique for Counting NATted Hosts", IMW'02 Nov. 6-8, 2002, Marseille, France, 2002.

[CPNI-TCP]
Gont, F., "Security Assessment of the Transmission Control Protocol (TCP)", United Kingdom's Centre for the Protection of National Infrastructure (CPNI) Technical Report, 2009, <https://www.gont.com.ar/papers/tn-03-09-security-assessment-TCP.pdf>.

[EFF] EFF, "Cover your tracks: See how trackers view your browser", 2020, <https://coveryourtracks.eff.org/>.

[FIPS-SHS] FIPS, "Secure Hash Standard (SHS)", Federal Information Processing Standards Publication 180-4, August 2015, <https://nvlpubs.nist.gov/nistpubs/FIPS/NIST.FIPS.180-4.pdf>.

[Fyodor1998] Fyodor, "Remote OS Detection via TCP/IP Stack Fingerprinting", Phrack Magazine, Volume 9, Issue 54, 1998, <http://www.phrack.org/archives/issues/54/9.txt>.

[Fyodor2004] Fyodor, "Idle scanning and related IP ID games", 2004, <http://www.insecure.org/nmap/idlescan.html>.

[Fyodor2006] Fyodor, "Remote OS Detection via TCP/IP Fingerprinting (2nd Generation)", 1998, <http://insecure.org/nmap/osdetect/>.

[I-D.gont-numeric-ids-sec-considerations] Gont, F. and I. Arce, "Security Considerations for Transient Numeric Identifiers Employed in Network Protocols", draft-gont-numeric-ids-sec-considerations-06 (work in progress), December 2020.

[I-D.irtf-pearg-numeric-ids-history] Gont, F. and I. Arce, "Unfortunate History of Transient Numeric Identifiers", draft-irtf-pearg-numeric-ids-history-04 (work in progress), December 2020.

[IANA-PROT] IANA, "Protocol Registries", <https://www.iana.org/protocols>.

[IPID-DEV] Klein, A. and B. Pinkas, "From IP ID to Device ID and KASLR Bypass (Extended Version)", June 2019, <https://arxiv.org/pdf/1906.10478.pdf>.

[Joncheray1995] Joncheray, L., "A Simple Active Attack Against TCP", Proc. Fifth Usenix UNIX Security Symposium, 1995.

[KASLR] PaX Team, "Address Space Layout Randomization", <https://pax.grsecurity.net/docs/aslr.txt>.

[Klein2007] Klein, A., "OpenBSD DNS Cache Poisoning and Multiple O/S Predictable IP ID Vulnerability", 2007, <http://www.trusteer.com/files/OpenBSD_DNS_Cache_Poisoning_and_Multiple_OS_Predictable_IP_ID_Vulnerability.pdf>.

[Morris1985] Morris, R., "A Weakness in the 4.2BSD UNIX TCP/IP Software", CSTR 117, AT&T Bell Laboratories, Murray Hill, NJ, 1985, <https://pdos.csail.mit.edu/~rtm/papers/117.pdf>.

[nmap] Fyodor, "Nmap: Free Security Scanner For Network Exploration and Audit", 2020, <https://www.insecure.org/nmap>.

[RFC4963] Heffner, J., Mathis, M., and B. Chandler, "IPv4 Reassembly Errors at High Data Rates", RFC 4963, DOI 10.17487/RFC4963, July 2007, <https://www.rfc-editor.org/info/rfc4963>.

[RFC6973] Cooper, A., Tschofenig, H., Aboba, B., Peterson, J., Morris, J., Hansen, M., and R. Smith, "Privacy Considerations for Internet Protocols", RFC 6973, DOI 10.17487/RFC6973, July 2013, <https://www.rfc-editor.org/info/rfc6973>.

[RFC7098] Carpenter, B., Jiang, S., and W. Tarreau, "Using the IPv6 Flow Label for Load Balancing in Server Farms", RFC 7098, DOI 10.17487/RFC7098, January 2014, <https://www.rfc-editor.org/info/rfc7098>.

[RFC7258] Farrell, S. and H. Tschofenig, "Pervasive Monitoring Is an Attack", BCP 188, RFC 7258, DOI 10.17487/RFC7258, May 2014, <https://www.rfc-editor.org/info/rfc7258>.

[RFC7707] Gont, F. and T. Chown, "Network Reconnaissance in IPv6 Networks", RFC 7707, DOI 10.17487/RFC7707, March 2016, <https://www.rfc-editor.org/info/rfc7707>.

[RFC7721] Cooper, A., Gont, F., and D. Thaler, "Security and Privacy Considerations for IPv6 Address Generation Mechanisms", RFC 7721, DOI 10.17487/RFC7721, March 2016, <https://www.rfc-editor.org/info/rfc7721>.

[RFC7739] Gont, F., "Security Implications of Predictable Fragment Identification Values", RFC 7739, DOI 10.17487/RFC7739, February 2016, <https://www.rfc-editor.org/info/rfc7739>.

[Sanfilippo1998a] Sanfilippo, S., "about the ip header id", Post to Bugtraq mailing-list, Mon Dec 14 1998, <http://seclists.org/bugtraq/1998/Dec/48>.

[Sanfilippo1998b] Sanfilippo, S., "Idle scan", Post to Bugtraq mailing-list, 1998, <https://github.com/antirez/hping/blob/master/docs/SPOOFED_SCAN.txt>.

[Sanfilippo1999] Sanfilippo, S., "more ip id", Post to Bugtraq mailing-list, 1999, <https://github.com/antirez/hping/raw/master/docs/MORE-FUN-WITH-IPID>.

[Shimomura1995] Shimomura, T., "Technical details of the attack described by Markoff in NYT", Message posted in USENET's comp.security.misc newsgroup Message-ID: <3g5gkl$5j1@ariel.sdsc.edu>, 1995, <https://www.gont.com.ar/docs/post-shimomura-usenet.txt>.

[Silbersack2005] Silbersack, M., "Improving TCP/IP security through randomization without sacrificing interoperability", EuroBSDCon 2005 Conference, 2005, <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.91.4542&rep=rep1&type=pdf>.

[TBIT] TBIT, "TBIT, the TCP Behavior Inference Tool", 2001, <http://www.icir.org/tbit/>.

[TCPT-uptime] McDanel, B., "TCP Timestamping - Obtaining System Uptime Remotely", March 2001, <https://securiteam.com/securitynews/5np0c153pi/>.

[Zalewski2001] Zalewski, M., "Strange Attractors and TCP/IP Sequence Number Analysis", 2001, <http://lcamtuf.coredump.cx/oldtcp/tcpseq.html>.

[Zalewski2002]
Zalewski, M., "Strange Attractors and TCP/IP Sequence Number Analysis - One Year Later", 2001, <http://lcamtuf.coredump.cx/newtcp/>.

[Zalewski2012]
Zalewski, M., "p0f v3 (version 3.09b)", 2012, <http://lcamtuf.coredump.cx/p0f.shtml>.

Appendix A. Algorithms and Techniques with Known Negative Implications

The following subsections document algorithms and techniques that generally have negative security and privacy implications.

A.1. Predictable Linear Identifiers Algorithm

One of the most trivial ways to achieve uniqueness with a low identifier reuse frequency is to produce a linear sequence. This type of algorithm has been employed in the past to generate identifiers of Categories #1, #2, and #4.

For example, the following algorithm has been employed (see e.g. [Morris1985], [Shimomura1995], [Silbersack2005] and [CPNI-TCP]) in a number of operating systems for selecting IP fragment IDs, TCP ephemeral ports, etc.:

```
      /* Initialization code */

      next_id = min_id;
      id_inc= 1;


      /* Transient Numeric ID selection function */

      id_range = max_id - min_id + 1;
      retry = id_range;

      do {
          if (next_id == max_id) {
              next_id = min_id;
          }
          else {
              next_id = next_id + id_inc;
          }

          if (check_suitable_id(next_id)) {
              return next_id;
          }

          retry--;

      } while (retry > 0);

      return ERROR;
```

Note:
check_suitable_id() is a function that checks whether the resulting identifier is acceptable (e.g., whether it's in use, etc.).

For obvious reasons, this algorithm results in predicable sequences. If a global counter is used (such as "next_id" in the example above), a node that learns one numeric identifier can guess past numeric identifiers and also predict future values to be generated by the same algorithm in the future. Since the value employed for the increments is known (such as "1" in this case), an attacker can sample two values, and learn the number of identifiers that have been were generated in-between. Furthermore, if the counter is initialized e.g. when the system its bootstrapped to some known value, the algorithm will leak additional information, such as the number of transmitted in the case of an IP ID generator [Sanfilippo1998a], or the system uptime in the case of TCP timestamps [TCPT-uptime].

A.2. Random-Increments Algorithm

This algorithm offers a middle ground between the algorithms that select random numeric identifiers (such as those described in Section 7.1.1 and Section 7.1.2), and those that select numeric identifiers as a monotonically-increasing function with a random origin (such as those described in Section 7.4.2 and Section 7.4.3).

```
    /* Initialization code */

    next_id = random();        /* Initialization value */
    id_rinc = 500;             /* Determines the trade-off */


    /* Transient Numeric ID selection function */

    id_range = max_id - min_id + 1;
    retry = id_range;


    do {
        /* Random increment */
        id_inc = (random() % id_rinc) + 1;

        if ( (max_id - next_id) >= id_inc){
            next_id = next_id + id_inc;
        }
        else {
            next_id = min_id + id_inc - (max_id - next_id);
        }

        if (check_suitable_id(next_id)) {
           return next_id;
        }

        retry = retry - id_inc;

    } while (retry > 0);

    return ERROR;
```

This algorithm aims at producing a global monotonically-increasing sequence of numeric identifiers, while avoiding the use of fixed increments, which would lead to trivially predictable sequences. The value "id_inc" allows for direct control of the trade-off between unpredictability and identifier reuse frequency. The smaller the value of "id_inc", the more similar this algorithm is to a

predicable, global linear ID generation algorithm. The larger the value of "id_inc", the more similar this algorithm is to the algorithm described in Section 7.1.1 of this document.

When the identifiers wrap, there is the risk of collisions of identifiers (i.e., identifier reuse). Therefore, "id_inc" should be selected according to the following criteria:

- It should maximize the wrapping time of the identifier space.
- It should minimize identifier reuse frequency.
- It should maximize unpredictability.

Clearly, these are competing goals, and the decision of which value of "id_inc" to use is a trade-off. Therefore, the value of "id_inc" should be configurable so that system administrators can make the trade-off for themselves. We note that the alternative algorithms discussed throughout this document offer better interoperability, security and privacy implications than this algorithm, and hence implementation of this algorithm is discouraged.

A.3. Re-using identifiers across different contexts

Employing the same identifier across contexts in which stability is not required (overloading the numeric identifier) usually has has negative security and privacy implications.

For example, in order to generate transient numeric identifiers of Category #2 or Category #3, an implementation or specification might be tempted to employ a source for the numeric identifiers which is known to provide unique values, but that may also have other properties such as being predictable or leaking information about the entity generating the identifier. This technique has been employed in the past for e.g. generating IPv6 IIDs. However, as noted in [RFC7721] and [RFC7707], embedding link-layer addresses in IPv6 IIDs not only results in predictable values, but also leaks information about the manufacturer of the underlying network interface card, allows for network activity correlation, and makes address-based scanning attacks feasible.

Authors' Addresses

Fernando Gont
SI6 Networks
Evaristo Carriego 2644
Haedo, Provincia de Buenos Aires 1706
Argentina

Email: fgont@si6networks.com
URI: https://www.si6networks.com

Ivan Arce
Quarkslab

Email: iarce@quarkslab.com
URI: https://www.quarkslab.com