Internet Engineering Task Force (IETF)
Request for Comments: 5879
Category: Informational
ISSN: 2070-1721

T. Kivinen
AuthenTec, Inc.
D. McDonald
Oracle Corporation

May 2010

Heuristics for Detecting ESP-NULL Packets

Abstract

This document describes a set of heuristics for distinguishing IPsec ESP-NULL (Encapsulating Security Payload without encryption) packets from encrypted ESP packets. These heuristics can be used on intermediate devices, like traffic analyzers, and deep-inspection engines, to quickly decide whether or not a given packet flow is encrypted, i.e., whether or not it can be inspected. Use of these heuristics does not require any changes made on existing IPsec hosts that are compliant with RFC 4303.

Status of This Memo

Copyright Notice

Table of Contents

1. Introduction

The ESP (Encapsulating Security Payload [RFC4303]) protocol can be used with NULL encryption [RFC2410] to provide authentication, integrity protection, and optionally replay detection, but without confidentiality. ESP without encryption (referred to as ESP-NULL) offers similar properties to IPsec’s AH (Authentication Header [RFC4302]). One reason to use ESP-NULL instead of AH is that AH cannot be used if there are NAT (Network Address Translation) devices on the path. With AH, it would be easy to detect packets that have only authentication and integrity protection, as AH has its own protocol number and deterministic packet length. With ESP-NULL, such detection is nondeterministic, in spite of the base ESP packet format being fixed.

In some cases, intermediate devices would like to detect ESP-NULL packets so they could perform deep inspection or enforce access control. This kind of deep inspection includes virus detection, spam filtering, and intrusion detection. As end nodes might be able to bypass those checks by using encrypted ESP instead of ESP-NULL, these kinds of scenarios also require very specific policies to forbid such circumvention.

These sorts of policy requirements usually mean that the whole network needs to be controlled, i.e., under the same administrative domain. Such setups are usually limited to inside the network of one enterprise or organization, and encryption is not used as the network is considered safe enough from eavesdroppers.

Because the traffic inspected is usually host-to-host traffic inside one organization, that usually means transport mode IPsec is used. Note, that most of the current uses of IPsec are not host-to-host traffic inside one organization, but for the intended use cases for the heuristics, this will most likely be the case. Also, the tunnel mode case is much easier to solve than transport mode as it is much easier to detect the IP header inside the ESP-NULL packet.

It should also be noted that even if new protocol modifications for ESP support easier detection of ESP-NULL in the future, this document will aid in the transition of older end-systems. That way, a solution can be implemented immediately, and not after 5-10 years of upgrade and deployment. Even with protocol modification for end nodes, the intermediate devices will need heuristics until they can assume that those protocol modifications can be found from all the end devices. To make sure that any solution does not break in the future, it would be best if such heuristics are documented -- i.e., publishing an RFC for what to do now, even though there might be a new protocol coming in the future that will solve the same problem in a better way.

1.1. Applicability: Heuristic Traffic Inspection and Wrapped ESP

There are two ways to enable intermediate security devices to distinguish between encrypted and unencrypted ESP traffic:

- The heuristics approach has the intermediate node inspect the unchanged ESP traffic, to determine with extremely high probability whether or not the traffic stream is encrypted.

- The Wrapped ESP (WESP) approach [RFC5840], in contrast, requires the ESP endpoints to be modified to support the new protocol. WESP allows the intermediate node to distinguish encrypted and unencrypted traffic deterministically, using a simpler implementation for the intermediate node.

Both approaches are being documented simultaneously by the IPsecME Working Group, with WESP being put on Standards Track while the heuristics approach is being published as an Informational RFC. While endpoints are being modified to adopt WESP, both approaches will likely coexist for years, because the heuristic approach is needed to inspect traffic where at least one of the endpoints has not been modified. In other words, intermediate nodes are expected to support both approaches in order to achieve good security and performance during the transition period.

1.2. Terminology

This document uses following terminology:

Flow

A TCP/UDP or IPsec flow is a stream of packets that are part of the same TCP/UDP or IPsec stream, i.e., TCP or UDP flow is a stream of packets having same 5 tuple (source and destination IP and port, and TCP/UDP protocol). Note, that this kind of flow is also called microflow in some documents.

Flow Cache

deep-inspection engines and similar devices use a cache of flows going through the device, and that cache keeps state of all flows going through the device.

IPsec Flow

An IPsec flow is a stream of packets sharing the same source IP, destination IP, protocol (ESP/AH), and Security Parameter Index (SPI). Strictly speaking, the source IP does not need to be a part of the flow identification, but it can be. For this reason, it is safer to assume that the source IP is always part of the flow identification.

2. Other Options

This document will discuss the heuristic approach of detecting ESP-NULL packets. There are some other options that can be used, and this section will briefly discuss them.

2.1. AH

The most logical approach would use the already defined protocol that offers authentication and integrity protection, but not confidentiality, namely AH. AH traffic is clearly marked as not encrypted, and can always be inspected by intermediate devices.

Using AH has two problems. First, as it also protects the IP headers, it will also protect against NATs on the path; thus, it will not work if there is a NAT on the path between end nodes. In some environments this might not be a problem, but some environments, include heavy use of NATs even inside the internal network of the enterprise or organization. NAT-Traversal (NAT-T, [RFC3948]) could be extended to support AH also, and the early versions of the NAT-T proposals did include that, but it was left out as it was not seen as necessary.

Another problem is that in the new IPsec Architecture [RFC4301] the support for AH is now optional, meaning not all implementations support it. ESP-NULL has been defined to be mandatory to implement by "Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH)" [RFC4835].

AH also has quite complex processing rules compared to ESP when calculating the Integrity Check Value (ICV), including things like zeroing out mutable fields. Also, as AH is not as widely used as ESP, the AH support is not as well tested in the interoperability events.

2.2. Mandating by Policy

Another easy way to solve this problem is to mandate the use of ESP-NULL with common parameters within an entire organization. This either removes the need for heuristics (if no ESP-encrypted traffic is allowed at all) or simplifies them considerably (only one set of parameters needs to be inspected, e.g., everybody in the organization who is using ESP-NULL must use HMAC-SHA-1-96 as their integrity algorithm). This does work unless one of a pair of communicating machines is not under the same administrative domain as the deep-inspection engine. (IPsec Security Associations (SAs) must be satisfactory to all communicating parties, so only one communicating peer needs to have a sufficiently narrow policy.) Also, such a solution might require some kind of centralized policy management to make sure everybody in an administrative domain uses the same policy, and that changes to that single policy can be coordinated throughout the administrative domain.

2.3. Modifying ESP

Several documents discuss ways of modifying ESP to offer intermediate devices information about an ESP packet’s use of NULL encryption. The following methods have been discussed: adding an IP-option, adding a new IP-protocol number plus an extra header [RFC5840], adding new IP-protocol numbers that tell the ESP-NULL parameters [AUTH-ONLY-ESP], reserving an SPI range for ESP-NULL [ESP-NULL], and using UDP encapsulation with a different format and ports.

All of the aforementioned documents require modification to ESP, which requires that all end nodes be modified before intermediate devices can assume that this new ESP format is in use. Updating end nodes will require a lot of time. An example of slow end-node deployment is Internet Key Exchange Protocol version 2 (IKEv2). Considering an implementation that requires both IKEv2 and a new ESP format, it would take several years, possibly as long as a decade, before widespread deployment.

3. Description of Heuristics

The heuristics to detect ESP-NULL packets will only require changes to those intermediate devices that do deep inspection or other operations that require the detection of ESP-NULL. As those nodes require changes regardless of any ESP-NULL method, updating intermediate nodes is unavoidable. Heuristics do not require updates or modifications to any other devices on the rest of the network, including (especially) end nodes.

In this document, it is assumed that an affected intermediate node will act as a stateful interception device, meaning it will keep state of the IPsec flows -- where flows are defined by the ESP SPI and IP addresses forming an IPsec SA -- going through it. The heuristics can also be used without storing any state, but performance will be worse in that case, as heuristic checks will need to be done for each packet, not only once per flow. This will also affect the reliability of the heuristics.

Generally, an intermediate node runs heuristics only for the first few packets of the new flow (i.e., the new IPsec SA). After those few packets, the node detects parameters of the IPsec flow, it skips detection heuristics, and it can perform direct packet-inspecting action based on its own policy. Once detected, ESP-NULL packets will never be detected as encrypted ESP packets, meaning that valid ESP-NULL packets will never bypass the deep inspection.

The only failure mode of these heuristics is to assume encrypted ESP packets are ESP-NULL packets, thus causing completely random packet data to be deeply inspected. An attacker can easily send random-looking ESP-NULL packets that will cause heuristics to detect packets as encrypted ESP, but that is no worse than sending non-ESP fuzz through an intermediate node. The only way an ESP-NULL flow can be mistaken for an encrypted ESP flow is if the ESP-NULL flow uses an authentication algorithm of which the packet inspector has no knowledge.

For hardware implementations, all the flow lookup based on the ESP next header number (50), source address, destination address, and SPI can be done by the hardware (there is usually already similar functionality there, for TCP/UDP flows). The heuristics can be implemented by the hardware, but using software will allow faster updates when new protocol modifications come out or new protocols need support.

As described in Section 7, UDP-encapsulated ESP traffic may also have Network Address Port Translation (NAPT) applied to it, and so there is already a 5-tuple state in the stateful inspection gateway.

4. IPsec Flows

ESP is a stateful protocol, meaning there is state stored in both end nodes of the ESP IPsec SA, and the state is identified by the pair of destination IP and SPI. Also, end nodes often fix the source IP address in an SA unless the destination is a multicast group. Typically, most (if not all) flows of interest to an intermediate device are unicast, so it is safer to assume the receiving node also uses a source address, and the intermediate device should therefore do the same. In some cases, this might cause extraneous cached ESP IPsec SA flows, but by using the source address, two distinct flows will never be mixed. For sites that heavily use multicast, such traffic is deterministically identifiable (224.0.0.0/4 for IPv4 and ff00::0/8 for IPv6), and an implementation can save the space of multiple cache entries for a multicast flow by checking the destination address first.

When the intermediate device sees a new ESP IPsec flow, i.e., a new flow of ESP packets where the source address, destination address, and SPI number form a triplet that has not been cached, it will start the heuristics to detect whether or not this flow is ESP-NULL. These heuristics appear in Section 8.

When the heuristics finish, they will label the flow as either encrypted (which tells that packets in this flow are encrypted, and cannot be ESP-NULL packets) or as ESP-NULL. This information, along with the ESP-NULL parameters detected by the heuristics, is stored to a flow cache, which will be used in the future when processing packets of the same flow.

Both encrypted ESP and ESP-NULL flows are processed based on the local policy. In normal operation, encrypted ESP flows are passed through or dropped per local policy, and ESP-NULL flows are passed to the deep-inspection engine. Local policy will also be used to determine other packet-processing parameters. Local policy issues will be clearly marked in this document to ease implementation.

In some cases, the heuristics cannot determine the type of flow from a single packet; and in that case, it might need multiple packets before it can finish the process. In those cases, the heuristics return "unsure" status. In that case, the packet processed based on the local policy and flow cache is updated with "unsure" status. Local policy for "unsure" packets could range from dropping (which encourages end-node retransmission) to queuing (which may preserve delivery, at the cost of artificially inflating round-trip times if they are measured). When the next packet to the flow arrives, it is heuristically processed again, and the cached flow may continue to be "unsure", marked as ESP, or marked as an ESP-NULL flow.

There are several reasons why a single packet might not be enough to detect the type of flow. One of them is that the next header number was unknown, i.e., if heuristics do not know about the protocol for the packet, they cannot verify it has properly detected ESP-NULL parameters, even when the packet otherwise looks like ESP-NULL. If the packet does not look like ESP-NULL at all, then the encrypted ESP status can be returned quickly. As ESP-NULL heuristics need to know the same protocols as a deep-inspection device, an ESP-NULL instance of an unknown protocol can be handled the same way as a cleartext instance of the same unknown protocol.

5. Deep-Inspection Engine

A deep-inspection engine running on an intermediate node usually checks deeply into the packet and performs policy decisions based on the contents of the packet. The deep-inspection engine should be able to tell the difference between success, failure, and garbage. Success means that a packet was successfully checked with the deep-inspection engine, and it passed the checks and is allowed to be forwarded. Failure means that a packet was successfully checked, but the actual checks done indicated that packets should be dropped, i.e., the packet contained a virus, was a known attack, or something similar.

Garbage means that the packet’s protocol headers or other portions were unparseable. For the heuristics, it would be useful if the deep-inspection engine could differentiate the garbage and failure cases, as garbage cases can be used to detect certain error cases (e.g., where the ESP-NULL parameters are incorrect, or the flow is really an encrypted ESP flow, not an ESP-NULL flow).

If the deep-inspection engine only returns failure for all garbage packets in addition to real failure cases, then a system implementing the ESP-NULL heuristics cannot recover from error situations quickly.

6. Special and Error Cases

There is a small probability that an encrypted ESP packet (which looks like it contains completely random bytes) will have plausible bytes in expected locations, such that heuristics will detect the packet as an ESP-NULL packet instead of detecting that it is encrypted ESP packet. The actual probabilities will be computed later in this document. Such a packet will not cause problems, as the deep-inspection engine will most likely reject the packet and return that it is garbage. If the deep-inspection engine is rejecting a high number of packets as garbage, it might indicate an original ESP-NULL detection for the flow was wrong (i.e., an encrypted ESP flow was improperly detected as ESP-NULL). In that case, the cached flow should be invalidated and discovery should happen again.

Each ESP-NULL flow should also keep statistics about how many packets have been detected as garbage by deep inspection, how many have passed checks, or how many have failed checks with policy violations (i.e., failed because of actual inspection policy failures, not because the packet looked like garbage). If the number of garbage packets suddenly increases (e.g., most of the packets start to look like garbage according to the deep-inspection engine), it is possible the old ESP-NULL SA was replaced by an encrypted ESP SA with an identical SPI. If both ends use random SPI generation, this is a very unlikely situation (1 in 2^32), but it is possible that some nodes reuse SPI numbers (e.g., a 32-bit memory address of the SA descriptor); thus, this situation needs to be handled.

Actual limits for cache invalidation are local policy decisions. Sample invalidation policies include: 50% of packets marked as garbage within a second, or if a deep-inspection engine cannot differentiate between garbage and failure, failing more than 95% of packets in last 10 seconds. For implementations that do not distinguish between garbage and failure, failures should not be treated too quickly as an indication of SA reuse. Often, single packets cause state-related errors that block otherwise normal packets from passing.

7. UDP Encapsulation

The flow lookup code needs to detect UDP packets to or from port 4500 in addition to the ESP packets, and perform similar processing to them after skipping the UDP header. Port-translation by NAT often rewrites what was originally 4500 into a different value, which means each unique port pair constitutes a separate IPsec flow. That is, UDP-encapsulated IPsec flows are identified by the source and destination IP, source and destination port number, and SPI number. As devices might be using IKEv2 Mobility and Multihoming (MOBIKE) ([RFC4555]), that also means that the flow cache should be shared between the UDP encapsulated IPsec flows and non-encapsulated IPsec flows. As previously mentioned, differentiating between garbage and actual policy failures will help in proper detection immensely.

Because the checks are run for packets having just source port 4500 or packets having just destination port 4500, this might cause checks to be run for non-ESP traffic too. Some traffic may randomly use port 4500 for other reasons, especially if a port-translating NAT is involved. The UDP encapsulation processing should also be aware of that possibility.

8. Heuristic Checks

Normally, HMAC-SHA1-96 or HMAC-MD5-96 gives 1 out of 2^96 probability that a random packet will pass the Hashed Message Authentication Code (HMAC) test. This yields a 99.999999999999999999999999998% probability that an end node will correctly detect a random packet as being invalid. This means that it should be enough for an intermediate device to check around 96 bits from the input packet. By comparing them against known values for the packet, a deep-inspection engine gains more or less the same probability as that which an end node is using. This gives an upper limit of how many bits heuristics need to check -- there is no point of checking much more than that many bits (since that same probability is acceptable for the end node). In most of the cases, the intermediate device does not need probability that is that high, perhaps something around 32-64 bits is enough.

IPsec's ESP has a well-understood packet layout, but its variable-length fields reduce the ability of pure algorithmic matching to one requiring heuristics and assigning probabilities.

8.1. ESP-NULL Format

The ESP-NULL format is as follows:

```
        0                   1                   2                   3
        0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                Security Parameter Index (SPI)                 |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                      Sequence Number                          |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                    IV (optional)                              |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                    Payload Data (variable)                    |
       ~                                                               ~
       |                                                               |
       +               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |               |     Padding (0-255 bytes)                     |
       +-+-+-+-+-+-+-+-+               +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |                               |  Pad Length   | Next Header   |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
       |             Integrity Check Value (variable)                  |
       ~                                                               ~
       |                                                               |
       +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 1

The output of the heuristics should provide information about whether the packet is encrypted ESP or ESP-NULL. In case it is ESP-NULL, the heuristics should also provide the Integrity Check Value (ICV) field length and the Initialization Vector (IV) length.

The currently defined ESP authentication algorithms have 4 different lengths for the ICV field.

Different ICV lengths for different algorithm:

Algorithm	ICV Length
AUTH_HMAC_MD5_96	96
AUTH_HMAC_SHA1_96	96
AUTH_AES_XCBC_96	96
AUTH_AES_CMAC_96	96
AUTH_HMAC_SHA2_256_128	128
AUTH_HMAC_SHA2_384_192	192
AUTH_HMAC_SHA2_512_256	256

Figure 2

In addition to the ESP authentication algorithms listed above, there is also the encryption algorithm ENCR_NULL_AUTH_AES_GMAC, which does not provide confidentiality but provides authentication, just like ESP-NULL. This algorithm has an ICV Length of 128 bits, and it also requires 8 bytes of IV.

In addition to the ICV length, there are also two possible values for IV lengths: 0 bytes (default) and 8 bytes (for ENCR_NULL_AUTH_AES_GMAC). Detecting the IV length requires understanding the payload, i.e., the actual protocol data (meaning TCP, UDP, etc.). This is required to distinguish the optional IV from the actual protocol data. How well the IV can be distinguished from the actual protocol data depends on how the IV is generated. If the IV is generated using a method that generates random-looking data (i.e., encrypted counter, etc.) then distinguishing protocol data from the IV is quite easy. If an IV is a counter or similar non-random value, then there are more possibilities for error. If the protocol (also known as the, "next header") of the packet is one that is not supported by the heuristics, then detecting the IV length is impossible; thus, the heuristics cannot finish. In that case, the heuristics return "unsure" and require further packets.

This document does not cover RSA authentication in ESP ([RFC4359]), as it is considered beyond the scope of this document.

8.2. Self Describing Padding Check

Before obtaining the next header field, the ICV length must be measured. Four different ICV lengths lead to four possible places for the pad length and padding. Implementations must be careful when trying larger sizes of the ICV such that the inspected bytes do not belong to data that is not payload data. For example, a 10-byte ICMP echo request will have zero-length padding, but any checks for 256-bit ICVs will inspect sequence number or SPI data if the packet actually contains a 96-bit or 128-bit ICV.

ICV lengths should always be checked from shortest to longest. It is much more likely to obtain valid-looking padding bytes in the cleartext part of the payload than from the ICV field of a longer ICV than what is currently inspected. For example, if a packet has a 96-bit ICV and the implementation starts checking for a 256-bit ICV first, it is possible that the cleartext part of the payload contains valid-looking bytes. If done in the other order, i.e., a packet having a 256-bit ICV and the implementation checks for a 96-bit ICV first, the inspected bytes are part of the longer ICV field, and should be indistinguishable from random noise.

Each ESP packet always has between 0-255 bytes of padding, and payload, pad length, and next header are always right aligned within a 4-byte boundary. Normally, implementations use a minimal amount of padding, but the heuristics method would be even more reliable if some extra padding is added. The actual padding data has bytes starting from 01 and ending at the pad length, i.e., exact padding and pad length bytes for 4 bytes of padding would be 01 02 03 04 04.

Two cases of ESP-NULL padding are matched bytes (like the 04 04 shown above), or the 0-byte padding case. In cases where there is one or more bytes of padding, a node can perform a very simple and fast test -- a sequence of N N in any of those four locations. Given four 2-byte locations (assuming the packet size allows all four possible ICV lengths), the upper-bound probability of finding a random encrypted packet that exhibits non-zero length ESP-NULL properties is:

1 - (1 - 255 / 65536) ^ 4 == 0.015 == 1.5%

In the cases where there are 0 bytes of padding, a random encrypted ESP packet has:

1 - (1 - 1 / 256) ^ 4 == 0.016 == 1.6%.

Together, both cases yield a 3.1% upper-bound chance of misclassifying an encrypted packet as an ESP-NULL packet.

In the matched bytes case, further inspection (counting the pad bytes backward and downward from the pad-length match) can reduce the number of misclassified packets further. A padding length of 255 means a specific 256^254 sequence of bytes must occur. This virtually eliminates pairs of ’FF FF’ as viable ESP-NULL padding.

Every one of the 255 pairs for padding length N has only a 1 / 256^N probability of being correct ESP-NULL padding. This shrinks the aforementioned 1.5% of matched pairs to virtually nothing.

At this point, a maximum of 1.6% of possible byte values remain, so the next header number is inspected. If the next header number is known (and supported), then the packet can be inspected based on the next header number. If the next header number is unknown (i.e., not any of those with protocol checking support) the packet is marked "unsure", because there is no way to detect the IV length without inspecting the inner protocol payload.

There are six different next header fields that are in common use (TCP (6), UDP (17), ICMP (1), Stream Control Transmission Protocol (SCTP) (132), IPv4 (4), and IPv6 (41)), and if IPv6 is in heavy use, that number increases to nine (Fragment (44), ICMPv6 (58), and IPv6 options (60)). To ensure that no packet is misinterpreted as an encrypted ESP packet even when it is an ESP-NULL packet, a packet cannot be marked as a failure even when the next header number is one of those that is not known and supported. In those cases, the packets are marked as "unsure".

An intermediate node’s policy, however, can aid in detecting an ESP-NULL flow even when the protocol is not a common-case one. By counting how many "unsure" returns obtained via heuristics, and after the receipt of a consistent, but unknown, next header number in same location (i.e., likely with the same ICV length), the node can conclude that the flow has high probability of being ESP-NULL (since it is unlikely that so many packets would pass the integrity check at the destination unless they are legitimate). The flow can be classified as ESP-NULL with a known ICV length but an unknown IV length.

Fortunately, in unknown protocol cases, the IV length does not matter. If the protocol is unknown to the heuristics, it will most likely be unknown by the deep-inspection engine also. It is therefore important that heuristics should support at least those same protocols as the deep-inspection engine. Upon receipt of any inner next header number that is known by the heuristics (and deep-inspection engine), the heuristics can detect the IV length properly.

8.3. Protocol Checks

Generic protocol checking is much easier with preexisting state. For example, when many TCP/UDP flows are established over one IPsec SA, a rekey produces a new SA that needs heuristics to detect its parameters, and those heuristics benefit from the existing TCP/UDP flows that were present in the previous IPsec SA. In that case, it is just enough to check that if a new IPsec SA has packets belonging to the flows of some other IPsec SA (previous IPsec SA before rekey), and if those flows are already known by the deep-inspection engine, it will give a strong indication that the new SA is really ESP-NULL.

The worst case scenario is when an end node starts up communication, i.e., it does not have any previous flows through the device. Heuristics will run on the first few packets received from the end node. The later subsections mainly cover these start-up cases, as they are the most difficult.

In the protocol checks, there are two different types of checks. The first check is for packet validity, i.e., certain locations must contain specific values. For example, an inner IPv4 header of an IPv4 tunnel packet must have its 4-bit version number set to 4. If it does not, the packet is not valid, and can be marked as a failure. Other positions depending on ICV and IV lengths must also be checked, and if all of them are failures, then the packet is a failure. If any of the checks are "unsure", the packet is marked as such.

The second type of check is for variable, but easy-to-parse values. For example, the 4-bit header length field of an inner IPv4 packet. It has a fixed value (5) as long as there are no inner IPv4 options. If the header-length has that specific value, the number of known "good" bits increases. If it has some other value, the known "good" bit count stays the same. A local policy might include reaching a bit count that is over a threshold (for example, 96 bits), causing a packet to be marked as valid.

8.3.1. TCP Checks

When the first TCP packet is fed to the heuristics, it is most likely going to be the SYN packet of the new connection; thus, it will have less useful information than other later packets might have. The best valid packet checks include checking that header length and flags have valid values and checking source and destination port numbers, which in some cases can be used for heuristics (but in general they cannot be reliably distinguished from random numbers apart from some well-known ports like 25/80/110/143).

The most obvious field, TCP checksum, might not be usable, as it is possible that the packet has already transited a NAT box that changed the IP addresses but assumed any ESP payload was encrypted and did not fix the transport checksums with the new IP addresses. Thus, the IP numbers used in the checksum are wrong; thus, the checksum is wrong. If the checksum is correct, it can again be used to increase the valid bit count, but verifying checksums is a costly operation, thus skipping that check might be best unless there is hardware to help the calculation. Window size, urgent pointer, sequence number, and acknowledgment numbers can be used, but there is not one specific known value for them.

One good method of detection is that if a packet is dropped, then the next packet will most likely be a retransmission of the previous packet. Thus, if two packets are received with the same source and destination port numbers, and where sequence numbers are either the same or right after each other, then it’s likely a TCP packet has been correctly detected. This heuristic is most helpful when only one packet is outstanding. For example, if a TCP SYN packet is lost (or dropped because of policy), the next packet would always be a retransmission of the same TCP SYN packet.

Existing deep-inspection engines usually do very good TCP flow checking already, including flow tracking, verification of sequence numbers, and reconstruction of the whole TCP flow. Similar methods can be used here, but they are implementation dependent and not described here.

8.3.2. UDP Checks

UDP header has even more problems than the TCP header, as UDP has even less known data. The checksum has the same problem as the TCP checksum, due to NATs. The UDP length field might not match the overall packet length, as the sender is allowed to include TFC (traffic flow confidentiality; see Section 2.7 of "IP Encapsulating Security Payload" [RFC4303]) padding.

With UDP packets similar multiple packet methods can be used as with TCP, as UDP protocols usually include several packets using same port numbers going from one end node to another, thus receiving multiple packets having a known pair of UDP port numbers is good indication that the heuristics have passed.

Some UDP protocols also use identical source and destination port numbers; thus, that is also a good check.

8.3.3. ICMP Checks

As ICMP messages are usually sent as return packets for other packets, they are not very common packets to get as first packets for the SA, the ICMP ECHO_REQUEST message being a noteworthy exception. ICMP ECHO_REQUEST has a known type, code, identifier, and sequence number. The checksum, however, might be incorrect again because of NATs.

For ICMP error messages, the ICMP message contains part of the original IP packet inside. Then, the same rules that are used to detect IPv4/IPv6 tunnel checks can be used.

8.3.4. SCTP Checks

SCTP [RFC4960] has a self-contained checksum, which is computed over the SCTP payload and is not affected by NATs unless the NAT is SCTP-aware. Even more than the TCP and UDP checksums, the SCTP checksum is expensive, and may be prohibitive even for deep packet inspections.

SCTP chunks can be inspected to see if their lengths are consistent across the total length of the IP datagram, so long as TFC padding is not present.

8.3.5. IPv4 and IPv6 Tunnel Checks

In cases of tunneled traffic, the packet inside contains a full IPv4 or IPv6 packet. Many fields are usable. For IPv4, those fields include version, header length, total length (again TFC padding might confuse things there), protocol number, and 16-bit header checksum. In those cases, the intermediate device should give the decapsulated IP packet to the deep-inspection engine. IPv6 has fewer usable fields, but the version number, packet length (modulo TFC confusion), and next header all can be used by deep packet inspection.

If all traffic going through the intermediate device is either from or to certain address blocks (for example, either to or from the company intranet prefix), this can also be checked by the heuristics.

9. Security Considerations

Attackers can always bypass ESP-NULL deep packet inspection by using encrypted ESP (or some other encryption or tunneling method) instead, unless the intermediate node’s policy requires dropping of packets that it cannot inspect. Ultimately, the responsibility for performing deep inspection, or allowing intermediate nodes to perform deep inspection, must rest on the end nodes. That is, if a server allows encrypted connections also, then an attacker who wants to attack the server and wants to bypass a deep-inspection device in the middle, will use encrypted traffic. This means that the protection of the whole network is only as good as the policy enforcement and protection of the end node. One way to enforce deep inspection for all traffic, is to forbid encrypted ESP completely, in which case ESP-NULL detection is easier, as all packets must be ESP-NULL based on the policy (heuristics may still be needed to find out the IV and ICV lengths, unless further policy restrictions eliminate the ambiguities).

Section 3 discusses failure modes of the heuristics. An attacker can poison flows, tricking inspectors into ignoring legitimate ESP-NULL flows, but that is no worse than injecting fuzz.

Forcing the use of ESP-NULL everywhere inside the enterprise, so that accounting, logging, network monitoring, and intrusion detection all work, increases the risk of sending confidential information where eavesdroppers can see it.

10. References

10.1. Normative References

[RFC2410] Glenn, R. and S. Kent, "The NULL Encryption Algorithm and Its Use With IPsec", RFC 2410, November 1998.

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

[RFC4302] Kent, S., "IP Authentication Header", RFC 4302, December 2005.

[RFC4303] Kent, S., "IP Encapsulating Security Payload (ESP)", RFC 4303, December 2005.

10.2. Informative References

[AUTH-ONLY-ESP] Hoffman, P. and D. McGrew, "An Authentication-only Profile for ESP with an IP Protocol Identifier", Work in Progress, August 2007.

[ESP-NULL] Bhatia, M., "Identifying ESP-NULL Packets", Work in Progress, December 2008.

[RFC3948] Huttunen, A., Swander, B., Volpe, V., DiBurro, L., and M. Stenberg, "UDP Encapsulation of IPsec ESP Packets", RFC 3948, January 2005.

[RFC4359] Weis, B., "The Use of RSA/SHA-1 Signatures within Encapsulating Security Payload (ESP) and Authentication Header (AH)", RFC 4359, January 2006.

[RFC4555] Eronen, P., "IKEv2 Mobility and Multihoming Protocol (MOBIKE)", RFC 4555, June 2006.

[RFC4835] Manral, V., "Cryptographic Algorithm Implementation Requirements for Encapsulating Security Payload (ESP) and Authentication Header (AH)", RFC 4835, April 2007.

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

[RFC5840] Grewal, K., Montenegro, G., and M. Bhatia, "Wrapped Encapsulating Security Payload (ESP) for Traffic Visibility", RFC 5840, April 2010.

Appendix A. Example Pseudocode

This appendix is meant for the implementors. It does not include all the required checks, and this is just example pseudocode, so final implementation can be very different. It mostly lists things that need to be done, but implementations can optimize steps depending on their other parts. For example, implementation might combine heuristics and deep inspection tightly together.

A.1. Fastpath

The following example pseudocode show the fastpath part of the packet processing engine. This part is usually implemented in hardware.

```
////////////////////////////////////////////////////////////
// This pseudocode uses following variables:
//
// SPI_offset:    Number of bytes between start of protocol
//                data and SPI.  This is 0 for ESP and
//                8 for UDP-encapsulated ESP (i.e, skipping
//                UDP header).
//
// IV_len:        Length of the IV of the ESP-NULL packet.
//
// ICV_len:       Length of the ICV of the ESP-NULL packet.
//
// State:         State of the packet, i.e., ESP-NULL, ESP, or
//                unsure.
//
// Also following data is taken from the packet:
//
// IP_total_len:  Total IP packet length.
// IP_hdr_len:    Header length of IP packet in bytes.
// IP_Src_IP:     Source address of IP packet.
// IP_Dst_IP:     Destination address of IP packet.
//
// UDP_len:       Length of the UDP packet taken from UDP header.
// UDP_src_port:  Source port of UDP packet.
// UDP_dst_port:  Destination port of UDP packet.
//
// SPI:           SPI number from ESP packet.
//
// Protocol:      Actual protocol number of the protocol inside
//                ESP-NULL packet.
// Protocol_off:  Calculated offset to the protocol payload data
//                inside ESP-NULL packet.
```

```
////////////////////////////////////////////////////////////
// This is the main processing code for the packet
// This will check if the packet requires ESP processing,
//
Process packet:
  * If IP protocol is ESP
       * Set SPI_offset to 0 bytes
       * Goto Process ESP
  * If IP protocol is UDP
       * Goto Process UDP
  * If IP protocol is WESP
       // For information about WESP processing, see WESP
       // specification.
       * Continue WESP processing
  * Continue Non-ESP processing

////////////////////////////////////////////////////////////
// This code is run for UDP packets, and it checks if the
// packet is UDP encapsulated UDP packet, or UDP
// encapsulated IKE packet, or keepalive packet.
//
Process UDP:
  // Reassembly is not mandatory here, we could
  // do reassembly also only after detecting the
  // packet being UDP encapsulated ESP packet, but
  // that would complicate the pseudocode here
  // a lot, as then we would need to add code
  // for checking whether or not the UDP header is in this
  // packet.
  // Reassembly is to simplify things
  * If packet is fragment
       * Do full reassembly before processing
  * If UDP_src_port != 4500 and UDP_dst_port != 4500
       * Continue Non-ESP processing
  * Set SPI_offset to 8 bytes
  * If UDP_len > 4 and first 4 bytes of UDP packet are 0x000000
       * Continue Non-ESP processing (pass IKE-packet)
  * If UDP_len > 4 and first 4 bytes of UDP packet are 0x000002
       * Continue WESP processing
  * If UDP_len == 1 and first byte is 0xff
       * Continue Non-ESP processing (pass NAT-Keepalive Packet)
  * Goto Process ESP
```

```
////////////////////////////////////////////////////////////
// This code is run for ESP packets (or UDP-encapsulated ESP
// packets).  This checks if IPsec flow is known, and
// if not calls heuristics.  If the IPsec flow is known
// then it continues processing based on the policy.
//
Process ESP:
  * If packet is fragment
       * Do full reassembly before processing
  * If IP_total_len < IP_hdr_len + SPI_offset + 4
       // If this packet was UDP encapsulated ESP packet then
       // this might be valid UDP packet that might
       // be passed or dropped depending on policy.
       * Continue normal packet processing
  * Load SPI from IP_hdr_len + SPI_offset
  * Initialize State to ESP
  // In case this was UDP encapsulated ESP, use UDP_src_port and
  // UDP_dst_port also when finding data from SPI cache.
  * Find IP_Src_IP + IP_Dst_IP + SPI from SPI cache
  * If SPI found
       * Load State, IV_len, ICV_len from cache
  * If SPI not found or State is unsure
       * Call Autodetect ESP parameters (drop to slowpath)
  * If State is ESP
       * Continue Non-ESP-NULL processing
  * Goto Check ESP-NULL packet

////////////////////////////////////////////////////////////
// This code is run for ESP-NULL packets, and this
// finds out the data required for deep-inspection
// engine (protocol number, and offset to data)
// and calls the deep-inspection engine.
//
Check ESP-NULL packet:
  * If IP_total_len < IP_hdr_len + SPI_offset + IV_len + ICV_len
                 + 4 (spi) + 4 (seq no) + 4 (protocol + padding)
       // This packet was detected earlier as being part of
       // ESP-NULL flow, so this means that either ESP-NULL
       // was replaced with other flow or this is an invalid packet.
       // Either drop or pass the packet, or restart
       // heuristics based on the policy
       * Continue packet processing
  * Load Protocol from IP_total_len - ICV_len - 1
  * Set Protocol_off to
        IP_hdr_len + SPI_offset + IV_len + 4 (spi) + 4 (seq no)
  * Do normal deep inspection on packet.
```

Figure 3

A.2. Slowpath

The following example pseudocode shows the actual heuristics part of the packet processing engine. This part is usually implemented in software.

```
////////////////////////////////////////////////////////////
// This pseudocode uses following variables:
//
// SPI_offset, IV_len, ICV_len, State, SPI,
// IP_total_len, IP_hdr_len, IP_Src_IP, IP_Dst_IP
// as defined in fastpath pseudocode.
//
// Stored_Check_Bits:Number of bits we have successfully
//                   checked to contain acceptable values
//                   in the actual payload data.  This value
//                   is stored/retrieved from SPI cache.
//
// Check_Bits:       Number of bits we have successfully
//                   checked to contain acceptable values
//                   in the actual payload data.  This value
//                   is updated during the packet
//                   verification.
//
// Last_Packet_Data: Contains selected pieces from the
//                   last packet.  This is used to compare
//                   certain fields of this packet to
//                   same fields in previous packet.
//
// Packet_Data:      Selected pieces of this packet, same
//                   fields as Last_Packet_Data, and this
//                   is stored as new Last_Packet_Data to
//                   SPI cache after this packet is processed.
//
// Test_ICV_len:     Temporary ICV length used during tests.
//                   This is stored to ICV_len when
//                   padding checks for the packet succeed
//                   and the packet didn’t yet have unsure
//                   status.
//
// Test_IV_len:      Temporary IV length used during tests.
//
// Pad_len:          Padding length from the ESP packet.
//
```

```
   // Protocol:          Protocol number of the packet inside ESP
   //                    packet.
   //
   // TCP.*:             Fields from TCP header (from inside ESP)
   // UDP.*:             Fields from UDP header (from inside ESP)

   ////////////////////////////////////////////////////////////
   // This code starts the actual heuristics.
   // During this the fastpath has already loaded
   // State, ICV_len, and IV_len in case they were
   // found from the SPI cache (i.e., in case the flow
   // had unsure status).
   //
   Autodetect ESP parameters:
     // First, we check if this is unsure flow, and
     // if so, we check next packet against the
     // already set IV/ICV_len combination.
     * If State is unsure
          * Call Verify next packet
          * If State is ESP-NULL
               * Goto Store ESP-NULL SPI cache info
          * If State is unsure
               * Goto Verify unsure
          // If we failed the test, i.e., State
          // was changed to ESP, we check other
          // ICV/IV_len values, i.e., fall through
     // ICV lengths are tested in order of ICV lengths,
     // from shortest to longest.
     * Call Try standard algorithms
     * If State is ESP-NULL
          * Goto Store ESP-NULL SPI cache info
     * Call Try 128bit algorithms
     * If State is ESP-NULL
          * Goto Store ESP-NULL SPI cache info
     * Call Try 192bit algorithms
     * If State is ESP-NULL
          * Goto Store ESP-NULL SPI cache info
     * Call Try 256bit algorithms
     * If State is ESP-NULL
          * Goto Store ESP-NULL SPI cache info
     // AUTH_DES_MAC and AUTH_KPDK_MD5 are left out from
     // this document.
     // If any of those test above set state to unsure
     // we mark IPsec flow as unsure.
     * If State is unsure
          * Goto Store unsure SPI cache info
```

```
    // All of the test failed, meaning the packet cannot
    // be ESP-NULL packet, thus we mark IPsec flow as ESP
    * Goto Store ESP SPI cache info
  ////////////////////////////////////////////////////////////
  // Store ESP-NULL status to the IPsec flow cache.
  //
  Store ESP-NULL SPI cache info:
    * Store State, IV_len, ICV_len to SPI cache
            using IP_Src_IP + IP_Dst_IP + SPI as key
    * Continue Check ESP-NULL packet

  ////////////////////////////////////////////////////////////
  // Store encrypted ESP status to the IPsec flow cache.
  //
  Store ESP SPI cache info:
    * Store State, IV_len, ICV_len to SPI cache
            using IP_Src_IP + IP_Dst_IP + SPI as key
    * Continue Check non-ESP-NULL packet

  ////////////////////////////////////////////////////////////
  // Store unsure flow status to IPsec flow cache.
  // Here we also store the Check_Bits.
  //
  Store unsure SPI cache info:
    * Store State, IV_len, ICV_len,
            Stored_Check_Bits to SPI cache
            using IP_Src_IP + IP_Dst_IP + SPI as key
    * Continue Check unknown packet

  ////////////////////////////////////////////////////////////
  // Verify this packet against the previously selected
  // ICV_len and IV_len values.  This will either
  // fail (and set state to ESP to mark we do not yet
  // know what type of flow this is) or will
  // increment Check_Bits.
  //
  Verify next packet:
    // We already have IV_len, ICV_len, and State loaded
    * Load Stored_Check_Bits, Last_Packet_Data from SPI Cache
    * Set Test_ICV_len to ICV_len, Test_IV_len to IV_len
    * Initialize Check_Bits to 0
    * Call Verify padding
    * If verify padding returned Failure
         // Initial guess was wrong, restart
         * Set State to ESP
         * Clear IV_len, ICV_len, State,
                 Stored_Check_Bits, Last_Packet_Data
                 from SPI Cache
```

```
          * Return
     // Ok, padding check succeeded again
     * Call Verify packet
     * If verify packet returned Failure
          // Guess was wrong, restart
          * Set State to ESP
          * Clear IV_len, ICV_len, State,
                  Stored_Check_Bits, Last_Packet_Data
                  from SPI Cache
          * Return
     // It succeeded and updated Check_Bits and Last_Packet_Data store
     // them to SPI cache.
     * Increment Stored_Check_Bits by Check_Bits
     * Store Stored_Check_Bits to SPI Cache
     * Store Packet_Data as Last_Packet_Data to SPI cache
     * Return

   ////////////////////////////////////////////////////////////
   // This will check if we have already seen enough bits
   // acceptable from the payload data, so we can decide
   // that this IPsec flow is ESP-NULL flow.
   //
   Verify unsure:
     // Check if we have enough check bits.
     * If Stored_Check_Bits > configured limit
          // We have checked enough bits, return ESP-NULL
          * Set State ESP-NULL
          * Goto Store ESP-NULL SPI cache info
     // Not yet enough bits, continue
     * Continue Check unknown packet

   ////////////////////////////////////////////////////////////
   // Check for standard 96-bit algorithms.
   //
   Try standard algorithms:
     // AUTH_HMAC_MD5_96, AUTH_HMAC_SHA1_96, AUTH_AES_XCBC_96,
     // AUTH_AES_CMAC_96
     * Set Test_ICV_len to 12, Test_IV_len to 0
     * Goto Check packet

   ////////////////////////////////////////////////////////////
   // Check for 128-bit algorithms, this is only one that
   // can have IV, so we need to check different IV_len values
   // here too.
   //
   Try 128bit algorithms:
     // AUTH_HMAC_SHA2_256_128, ENCR_NULL_AUTH_AES_GMAC
     * Set Test_ICV_len to 16, Test_IV_len to 0
```

```
    * If IP_total_len < IP_hdr_len + SPI_offset
         + Test_IV_len + Test_ICV_len
         + 4 (spi) + 4 (seq no) + 4 (protocol + padding)
         * Return
    * Call Verify padding
    * If verify padding returned Failure
         * Return
    * Initialize Check_Bits to 0
    * Call Verify packet
    * If verify packet returned Failure
         * Goto Try GMAC
    // Ok, packet seemed ok, but go now and check if we have enough
    // data bits so we can assume it is ESP-NULL
    * Goto Check if done for unsure

  ////////////////////////////////////////////////////////////
  // Check for GMAC MACs, i.e., MACs that have an 8-byte IV.
  //
  Try GMAC:
    // ENCR_NULL_AUTH_AES_GMAC
    * Set Test_IV_len to 8
    * If IP_total_len < IP_hdr_len + SPI_offset
         + Test_IV_len + Test_ICV_len
         + 4 (spi) + 4 (seq no) + 4 (protocol + padding)
         * Return
    * Initialize Check_Bits to 0
    * Call Verify packet
    * If verify packet returned Failure
         // Guess was wrong, continue
         * Return
    // Ok, packet seemed ok, but go now and check if we have enough
    // data bits so we can assume it is ESP-NULL
    * Goto Check if done for unsure

  ////////////////////////////////////////////////////////////
  // Check for 192-bit algorithms.
  //
  Try 192bit algorithms:
    // AUTH_HMAC_SHA2_384_192
    * Set Test_ICV_len to 24, Test_IV_len to 0
    * Goto Check packet

  ////////////////////////////////////////////////////////////
  // Check for 256-bit algorithms.
  //
  Try 256bit algorithms:
    // AUTH_HMAC_SHA2_512_256
    * Set Test_ICV_len to 32, Test_IV_len to 0
    * Goto Check packet

  ////////////////////////////////////////////////////////////
  // This actually does the checking for the packet, by
  // first verifying the length, and then self describing
  // padding, and if that succeeds, then checks the actual
  // payload content.
  //
  Check packet:
    * If IP_total_len < IP_hdr_len + SPI_offset
         + Test_IV_len + Test_ICV_len
         + 4 (spi) + 4 (seq no) + 4 (protocol + padding)
         * Return
    * Call Verify padding
    * If verify padding returned Failure
         * Return
    * Initialize Check_Bits to 0
    * Call Verify packet
    * If verify packet returned Failure
         // Guess was wrong, continue
         * Return
    // Ok, packet seemed ok, but go now and check if we have enough
    // data bits so we can assume it is ESP-NULL
    * Goto Check if done for unsure

  ////////////////////////////////////////////////////////////
  // This code checks if we have seen enough acceptable
  // values in the payload data, so we can decide that this
  // IPsec flow is ESP-NULL flow.
  //
  Check if done for unsure:
    * If Stored_Check_Bits > configured limit
         // We have checked enough bits, return ESP-NULL
         * Set State ESP-NULL
         * Set IV_len to Test_IV_len, ICV_len to Test_ICV_len
         * Clear Stored_Check_Bits, Last_Packet_Data from SPI Cache
         * Return
    // Not yet enough bits, check if this is first unsure, if so
    // store information.  In case there are multiple
    // tests succeeding, we always assume the first one
    // (the one using shortest MAC) is the one we want to
    // check in the future.
    * If State is not unsure
         * Set State unsure
         // These values will be stored to SPI cache if
         // the final state will be unsure
         * Set IV_len to Test_IV_len, ICV_len to Test_ICV_len
         * Set Stored_Check_Bits as Check_Bits
    * Return

  ////////////////////////////////////////////////////////////
  // Verify self describing padding
  //
  Verify padding:
    * Load Pad_len from IP_total_len - Test_ICV_len - 2
    * Verify padding bytes at
                 IP_total_len - Test_ICV_len - 1 - Pad_len ..
                 IP_total_len - Test_ICV_len - 2 are
                 1, 2, ..., Pad_len
    * If Verify of padding bytes succeeded
         * Return Success
    * Return Failure

  ////////////////////////////////////////////////////////////
  // This will verify the actual protocol content inside ESP
  // packet.
  //
  Verify packet:
    // We need to first check things that cannot be set, i.e., if any of
    // those are incorrect, then we return Failure.  For any
    / fields that might be correct, we increment the Check_Bits
    // for a suitable amount of bits.  If all checks pass, then
    // we just return Success, and the upper layer will then
    // later check if we have enough bits checked already.
    * Load Protocol From IP_total_len - Test_ICV_len - 1
    * If Protocol TCP
         * Goto Verify TCP
    * If Protocol UDP
         * Goto Verify UDP
    // Other protocols can be added here as needed, most likely same
    // protocols as deep inspection does.
    // Tunnel mode checks (protocol 4 for IPv4 and protocol 41 for
    // IPv6) is also left out from here to make the document shorter.
    * Return Failure

  ////////////////////////////////////////////////////////////
  // Verify TCP protocol headers
  //
  Verify TCP:
    // First we check things that must be set correctly.
    * If TCP.Data_Offset field < 5
        // TCP head length too small
        * Return Failure
    // After that, we start to check things that do not
    // have one definitive value, but can have multiple possible
    // valid values.
    * If TCP.ACK bit is not set, then check
         that TCP.Acknowledgment_number field contains 0
         // If the ACK bit is not set, then the acknowledgment
         // field usually contains 0, but I do not think
         // RFCs mandate it being zero, so we cannot make
         // this a failure if it is not so.
         * Increment Check_Bits by 32
    * If TCP.URG bit is not set, then check
         that TCP.Urgent_Pointer field contains 0
         // If the URG bit is not set, then urgent pointer
         // field usually contains 0, but I do not think
         // RFCs mandate it being zero, so we cannot make
         // this failure if it is not so.
         * Increment Check_Bits by 16
    * If TCP.Data_Offset field == 5
        * Increment Check_Bits by 4
    * If TCP.Data_Offset field > 5
        * If TCP options format is valid and it is padded correctly
             * Increment Check_Bits accordingly
        * If TCP options format was garbage
             * Return Failure
    * If TCP.checksum is correct
        // This might be wrong because packet passed NAT, so
        // we cannot make this failure case.
        * Increment Check_Bits by 16
    // We can also do normal deeper TCP inspection here, i.e.,
    // check that the SYN/ACK/FIN/RST bits are correct and state
    // matches the state of existing flow if this is packet
    // to existing flow, etc.
    // If there is anything clearly wrong in the packet (i.e.,
    // some data is set to something that it cannot be), then
    // this can return Failure; otherwise, it should just
    // increment Check_Bits matching the number of bits checked.
    //
    // We can also check things here compared to the last packet
    * If Last_Packet_Data.TCP.source port =
         Packet_Data.TCP.source_port and
         Last_Packet_Data.TCP.destination port =
         Packet_Data.TCP.destination port
         * Increment Check_Bits by 32
    * If Last_Packet_Data.TCP.Acknowledgement_number =
         Packet_Data.TCP.Acknowledgement_number
         * Increment Check_Bits by 32
    * If Last_Packet_Data.TCP.sequence_number =
         Packet_Data.TCP.sequence_number
         * Increment Check_Bits by 32
    // We can do other similar checks here
    * Return Success

  ////////////////////////////////////////////////////////////
  // Verify UDP protocol headers
  //
  Verify UDP:
    // First we check things that must be set correctly.
    * If UDP.UDP_length > IP_total_len - IP_hdr_len - SPI_offset
        - Test_IV_len - Test_ICV_len - 4 (spi)
        - 4 (seq no) - 1 (protocol)
        - Pad_len - 1 (Pad_len)
        * Return Failure
    * If UDP.UDP_length < 8
        * Return Failure
    // After that, we start to check things that do not
    // have one definitive value, but can have multiple possible
    // valid values.
    * If UDP.UDP_checksum is correct
        // This might be wrong because packet passed NAT, so
        // we cannot make this failure case.
        * Increment Check_Bits by 16
    * If UDP.UDP_length = IP_total_len - IP_hdr_len - SPI_offset
         - Test_IV_len - Test_ICV_len - 4 (spi)
         - 4 (seq no) - 1 (protocol)
         - Pad_len - 1 (Pad_len)
         // If there is no TFC padding then UDP_length
         // will be matching the full packet length
         * Increment Check_Bits by 16
    // We can also do normal deeper UDP inspection here.
    // If there is anything clearly wrong in the packet (i.e.,
    // some data is set to something that it cannot be), then
    // this can return Failure; otherwise, it should just
    // increment Check_Bits matching the number of bits checked.
    //
    // We can also check things here compared to the last packet
    * If Last_Packet_Data.UDP.source_port =
         Packet_Data.UDP.source_port and
         Last_Packet_Data.destination_port =
         Packet_Data.UDP.destination_port
         * Increment Check_Bits by 32
    * Return Success
```

Figure 4

Authors' Addresses

Tero Kivinen
AuthenTec, Inc.
Fredrikinkatu 47
Helsinki FIN-00100
FI

EMail: kivinen@iki.fi

Daniel L. McDonald
Oracle Corporation
35 Network Drive
MS UBUR02-212
Burlington, MA 01803
USA

EMail: danmcd@opensolaris.org