Multiple vulnerabilities in Tianocore's EDK2 UEFI implementation

This document describes several vulnerabilites in the IP stack of Tianocore's EDK2 open source UEFI implementation.

These vulnerabilities can be exploited by unauthenticated remote attackers on the same broadcast domain (local network) and, in some cases, by attackers on remote networks. They are exploitable on systems that have the PXE boot option enabled. Although this option is disabled by default, it is very common to see it enabled on server nodes in datacenters and HPC environments.

The impact of these vulnerabilities include denial of service, information leakage, remote code execution, DNS cache poisoning and network session hijacking. Exploitation of bugs 02, 06 and 07 for remote code execution is deemed straight forward as EDK2 does not officially employ mitigations such as stack cookies or address space layout randomization.

Affected vendors

- Tianocore EDK2 UEFI implementation
  - NetworkPkg PXE IP stack.

Other Vendors that use the EDK2 NetworkPkg module (non-exhaustive list):

- Google
  - ChromiumOS
- Arm
  - Arm reference solutions
- Insyde
  - Insyde H20 UEFI BIOS
- AMI
  - Aptio OpenEdition

Vulnerability details

Technical details about each vulnerability are provided below. Proof-of-concept programs to reproduce bugs 01 to 08 are available in a companion file.

Bug 01 - edk2/NetworkPkg: Out-of-bounds read when processing IA_NA/IA_TA options in a DHCPv6 Advertise message

When trying to boot using PXE over IPv6, the EDK2 firmware starts by sending DHCPv6 `Solicit` messages. DHCPv6 servers answer with an `Advertise` message. `Advertise` messages are handled in EDK2 by the `Dhcp6HandleAdvertiseMsg` function in [`NetworkPkg/Dhcp6Dxe/Dhcp6Io.c`], which calls `Dhcp6SeekStsOption` in order to search for `Status` codes in the received message. A `Status` code may appear as an option in the DHCPv6

message, or as an option inside another option, such as the `IA_NA` or `IA_TA` options.

When seeking for `Status` codes encapsulated in `IA_NA` or `IA_TA` options, the vulnerable function `Dhcp6SeekStsOption` in `NetworkPkg/Dhcp6Dxe/Dhcp6Io.c` does the following:

```
EFI_STATUS
Dhcp6SeekStsOption (
  IN     DHCP6_INSTANCE    *Instance,
  IN     EFI_DHCP6_PACKET  *Packet,
  OUT    UINT8             **Option
  )
{
[...]
    //
    // Seek in encapsulated options, IA_NA and IA_TA.
    //
    *Option = Dhcp6SeekIaOption (
                  Packet->Dhcp6.Option,
                  Packet->Length - sizeof (EFI_DHCP6_HEADER),
                  &Instance->Config->IaDescriptor
                  );
    if (*Option == NULL) {
        return EFI_SUCCESS;
    }

[...]
    if (Instance->Config->IaDescriptor.Type == Dhcp6OptIana) {
[1]     IaInnerOpt = *Option + 16;
[2]     IaInnerLen = (UINT16)(NTOHS (ReadUnaligned16 ((UINT16 *)(*Option + 2))) - 12);
    } else {
[3]     IaInnerOpt = *Option + 8;
[4]     IaInnerLen = (UINT16)(NTOHS (ReadUnaligned16 ((UINT16 *)(*Option + 2))) - 4);
    }

[5] *Option = Dhcp6SeekOption (IaInnerOpt, IaInnerLen, Dhcp6OptStatusCode);

[...]
```

The `IA_NA` and `IA_TA` options in an `Advertise` message are trusted without doing basic sanity checks, e.g.: - in the case of `IA_NA` options: - that the option is at least 16 bytes in length before setting `IaInnerOpt` to `*Option + 16`, at [1]. - that the value of the `optlen` field of the option is at least 12, before subtracting 12 from it, at [2].

```
- in the case of `IA_TA` options:
    - that the option is at least 8 bytes in length before setting `IaInnerOpt` to `*Option
```

- that the value of the `optlen` field of the option is at least 4, before subtracting 4

The lack of checking for sane values in the `optlen` field of these options allows to trigger an integer underflow, by setting the `optlen` field to a value < 12 (in the case of `IA_NA`) or < 4 (in the case of `IA_TA`).

As an example, if we send a DHCPv6 `Advertise` message containing an `IA_NA` option with its `optlen` field set to 15, when the code above calls `Dhcp6SeekOption` at [5] to scan forward for status options, the `IaInnerLen` parameter will be set to 0xFFFF (15 - 16), and as a result function `Dhcp6SeekOption` will attempt to read memory well past the end of the received packet.

Bug 02 - edk2/NetworkPkg: Buffer overflow in the DHCPv6 client via a long Server ID option

After doing the initial `Solicit`/`Advertise` exchange, a DHCPv6 client needs to send a `Request` message. This is done via the `Dhcp6SendRequestMsg` function in `NetworkPkg/Dhcp6Dxe/Dhcp6Io.c`.

This `Request` message needs to keep a few DHCPv6 options sent during the `Solicit` message, such as `OPTION_ORO` (0x6), `OPTION_CLIENT_ARCH_TYPE` (0x3D), `OPTION_NII` (0x3E), and `OPTION_VENDOR_CLASS` (0x10). These options are kept in `Instance->Config->OptionList`. Their lengths are added up, in order to calculate the size of the buffer needed to store the `Request` packet. Notice that the total size of the allocation is `DHCP6_BASE_PACKET_SIZE` (1024, defined in `NetworkPkg/Dhcp6Dxe/Dhcp6Impl.h`) plus the sum of the lengths of the options. In our tests `UserLen` typically ends up having a value of 67, so the total size of the allocation is 1091 bytes. In NetworkPkg/Dhcp6Dxe/Dhcp6Io.c

```
EFI_STATUS
Dhcp6SendRequestMsg (
    IN DHCP6_INSTANCE  *Instance
    )
    [...]
    //
    // Calculate the added length of customized option list.
    //
    UserLen = 0;
    for (Index = 0; Index < Instance->Config->OptionCount; Index++) {
        UserLen += (NTOHS (Instance->Config->OptionList[Index]->OpLen) + 4);
    }

    //
    // Create the Dhcp6 packet and initialize common fields.
    //
    Packet = AllocateZeroPool (DHCP6_BASE_PACKET_SIZE + UserLen);
    if (Packet == NULL) {
        return EFI_OUT_OF_RESOURCES;
```

```
    }

    Packet->Size                          = DHCP6_BASE_PACKET_SIZE + UserLen;
    [...]
```

The `Request` message also needs to retain the `OPTION_SERVERID` (0x2) option that was previously received from the DHCPv6 server in the `Advertise` message. It is retrieved this way:

```
EFI_STATUS
Dhcp6SendRequestMsg (
    IN DHCP6_INSTANCE  *Instance
    )
    [...]
    //
    // Get the server Id from the selected advertisement message.
    //
    Option = Dhcp6SeekOption (
             Instance->AdSelect->Dhcp6.Option,
             Instance->AdSelect->Length - 4,
             Dhcp6OptServerId
             );
    if (Option == NULL) {
        return EFI_DEVICE_ERROR;
    }

    ServerId = (EFI_DHCP6_DUID *)(Option + 2);
    [...]
```

Then, the `Request` packet is built by assembling all the needed options, one after the other: first the Client ID, then the Elapsed Time, then the Server ID... Notice that when appending the Server ID option (which is fully controlled by the DHCPv6 server), it's `Length` field is fully trusted, without any sanity checks. Therefore, a Server ID option with an overly large `Length` field can overflow the `Packet->Dhcp6.Option` buffer with fully controlled data (overflow data comes from the `DUID` field of the Server ID option).

```
  [...]
  //
  // Assembly Dhcp6 options for request message.
  //
  Cursor = Packet->Dhcp6.Option;

  Length = HTONS (ClientId->Length);
  Cursor = Dhcp6AppendOption (
             Cursor,
             HTONS (Dhcp6OptClientId),
             Length,
```

```
            ClientId->Duid
            );

  Cursor = Dhcp6AppendETOption (
            Cursor,
            Instance,
            &Elapsed
            );

  Cursor = Dhcp6AppendOption (
            Cursor,
            HTONS (Dhcp6OptServerId),
            ServerId->Length,
            ServerId->Duid
            );
```

There's an attempt at a sanity check in `Dhcp6SendRequestMsg`, but it's an `ASSERT`, which means that the check gets removed from release builds. Plus, it's a post-check, so even if it was present in release builds, it might be already too late.

```
  //
  // Determine the size/length of packet.
  //
  Packet->Length += (UINT32)(Cursor - Packet->Dhcp6.Option);
  ASSERT (Packet->Size > Packet->Length + 8);
```

The very same bug seems to be present in functions `Dhcp6SendDeclineMsg`, `Dhcp6SendReleaseMsg`, and `Dhcp6SendRenewRebindMsg` (`NetworkPkg/Dhcp6Dxe/Dhcp6Io.c`).

Bug 03 - edk2/NetworkPkg: Out-of-bounds read when handling a ND Redirect message with truncated options

Function `Ip6ProcessRedirect` in `NetworkPkg/Ip6Dxe/Ip6Nd.c` handles Neighbor Discovery protocol's `Redirect` messages. This function calls `Ip6IsNDOptionValid` (`NetworkPkg/Ip6Dxe/Ip6Option.c`) to verify that all the options included in the `Redirect` message are valid.

In `NetworkPkg/Ip6Dxe/Ip6Nd.c`:

```
EFI_STATUS
Ip6ProcessRedirect (
  IN IP6_SERVICE     *IpSb,
  IN EFI_IP6_HEADER  *Head,
  IN NET_BUF         *Packet
  )
{
    [...]
```

```
    // All included options have a length that is greater than zero.
    //
    OptionLen = (UINT16)(Head->PayloadLength - IP6_REDITECT_LENGTH);
    if (OptionLen != 0) {
        Option = NetbufGetByte (Packet, IP6_REDITECT_LENGTH, NULL);
        ASSERT (Option != NULL);

        if (!Ip6IsNDOptionValid (Option, OptionLen)) {
            goto Exit;
        }
    }
  [...]
```

If the options section of an incoming ND `Redirect` message is made of a single option, which is truncated and composed of only the Option Code (i.e. 1 single byte), `Ip6IsNDOptionValid` returns TRUE. This happens because `Ip6IsNDOptionValid` is called with parameter `OptionLen` == 1, and `sizeof(IP6_OPTION_HEADER)` is 2, therefore the `while` loop in charge of traversing the list of options ([1]) is never entered, and we just reach the `return TRUE` at the end ([2]), meaning that the truncated option is considered as valid, even though it is not:

```
BOOLEAN
Ip6IsNDOptionValid (
  IN UINT8   *Option,
  IN UINT16  OptionLen
  )
{
    Offset = 0;

    //
    // RFC 4861 states that Neighbor Discovery packet can contain zero or more
    // options. Start processing the options if at least Type + Length fields
    // fit within the input buffer.
    //
[1] while (Offset + sizeof (IP6_OPTION_HEADER) - 1 < OptionLen) {
        [...]
    }

[2] return TRUE;
}
```

After that, back in `Ip6ProcessRedirect`, options are actually processed. If our options section of the packet is composed of a single byte (i.e. a truncated option composed of just the opcode), the `Length` variable takes value 1 at [1], so it enters the `while` block at [2], and at [3] or [4] (depending on the opcode we specified in the truncated option), one byte is read past the end of the packet,

which is the missing length field of our truncated option.

In NetworkPkg/Ip6Dxe/Ip6Nd.c:

```
  // Check the options. The only interested option here is the target-link layer
  // address option.
  //
[1] Length          = Packet->TotalSize - 40;
    Option          = (UINT8 *)(IcmpDest + 1);
    LinkLayerOption = NULL;
[2] while (Length > 0) {
        switch (*Option) {
        case Ip6OptionEtherTarget:

            LinkLayerOption = (IP6_ETHER_ADDR_OPTION *)Option;
[3]         OptLen          = LinkLayerOption->Length;
            if (OptLen != 1) {
                //
                // For ethernet, the length must be 1.
                //
                goto Exit;
            }

            break;

        default:

[4]         OptLen = *(Option + 1);
            if (OptLen == 0) {
                //
                // A length of 0 is invalid.
                //
                goto Exit;
            }

            break;
    }

    Length -= 8 * OptLen;
    Option += 8 * OptLen;
  }
```

Bug 04 - edk2/NetworkPkg: Infinite loop when parsing unknown options in the Destination Options header

Function `Ip6IsExtsValid` (`NetworkPkg/Ip6Dxe/Ip6Option.c`) validates the extension headers that can be found in an incoming IPv6 packet. When dealing

with a DESTINATION_OPTIONS extension header, function Ip6IsOptionValid is called to validate whatever options are embedded in said DESTINATION_OPTIONS header.

In NetworkPkg/Ip6Dxe/Ip6Option.c:

```
BOOLEAN
Ip6IsExtsValid (
  IN IP6_SERVICE  *IpSb           OPTIONAL,
  IN NET_BUF      *Packet         OPTIONAL,
  IN UINT8        *NextHeader,
  IN UINT8        *ExtHdrs,
  IN UINT32       ExtHdrsLen,
  IN BOOLEAN      Rcvd,
  OUT UINT32      *FormerHeader   OPTIONAL,
  OUT UINT8       **LastHeader,
  OUT UINT32      *RealExtsLen    OPTIONAL,
  OUT UINT32      *UnFragmentLen  OPTIONAL,
  OUT BOOLEAN     *Fragmented     OPTIONAL
  )
{
  [...]
  while (Offset <= ExtHdrsLen) {
    switch (*NextHeader) {
    [...]
    case IP6_DESTINATION:
        [...]
        Offset++;
        Option    = ExtHdrs + Offset;
        OptionLen = (UINT8)((*Option + 1) * 8 - 2);
        Option++;
        Offset++;

        if ((IpSb != NULL) && (Packet != NULL) && !Ip6IsOptionValid (IpSb, Packet, Option, O
          return FALSE;
        }
        [...]
```

Function Ip6IsOptionValid handles different option types. If it's not one of Pad1, PadN or RouterAlert, the switch case falls into the default clause at [2]. In that case, the option code is masked with Ip6OptionMask (0xC0) at [3], and if the result of the AND operation is Ip6OptionSkip (0x00), then the Offset variable is updated by adding the value of the Length field of the option at [4]. Notice that it doesn't check if the Length field of the option is 0, in which case the Offset value is never modified, and thus an infinite loop ensues, since execution can never break out of the while loop at [1].

In NetworkPkg/Ip6Dxe/Ip6Option.c:

```
Ip6IsOptionValid (
  IN IP6_SERVICE  *IpSb,
  IN NET_BUF      *Packet,
  IN UINT8        *Option,
  IN UINT8        OptionLen,
  IN UINT32       Pointer
  )
{
    [...]
[1]  while (Offset < OptionLen) {
        OptionType = *(Option + Offset);

        switch (OptionType) {
        case Ip6OptionPad1:
            [...]
            break;
        case Ip6OptionPadN:
            [...]
            break;
        case Ip6OptionRouterAlert:
            [...]
            break;
        [...]
[2]     default:
            //
            // The highest-order two bits specify the action must be taken if
            // the processing IPv6 node does not recognize the option type.
            //
[3]         switch (OptionType & Ip6OptionMask) {
            case Ip6OptionSkip:
[4]             Offset = (UINT8)(Offset + *(Option + Offset + 1));
                break;
            [...]
```

Bug 05 - edk2/NetworkPkg: Infinite loop when parsing a PadN option in the Destination Options header

Function `Ip6IsExtsValid` (`NetworkPkg/Ip6Dxe/Ip6Option.c`) validates the extension headers that can be found in an incoming IPv6 packet. When dealing with a `DESTINATION_OPTIONS` extension header, function `Ip6IsOptionValid` is called to validate whatever options are embedded in said `DESTINATION_OPTIONS` header:

```
BOOLEAN
Ip6IsExtsValid (
```

```
  IN IP6_SERVICE  *IpSb           OPTIONAL,
  IN NET_BUF      *Packet         OPTIONAL,
  IN UINT8        *NextHeader,
  IN UINT8        *ExtHdrs,
  IN UINT32       ExtHdrsLen,
  IN BOOLEAN      Rcvd,
  OUT UINT32      *FormerHeader   OPTIONAL,
  OUT UINT8       **LastHeader,
  OUT UINT32      *RealExtsLen    OPTIONAL,
  OUT UINT32      *UnFragmentLen  OPTIONAL,
  OUT BOOLEAN     *Fragmented     OPTIONAL
  )
{
  [...]
  while (Offset <= ExtHdrsLen) {
    switch (*NextHeader) {
    [...]
    case IP6_DESTINATION:
        [...]
        Offset++;
        Option    = ExtHdrs + Offset;
        OptionLen = (UINT8)((*Option + 1) * 8 - 2);
        Option++;
        Offset++;

        if ((IpSb != NULL) && (Packet != NULL) && !Ip6IsOptionValid (IpSb, Packet, Option, O
          return FALSE;
        }
        [...]
```

Function `Ip6IsOptionValid` handles different option types. One of those supported types is `PadN` ([2]). In that case, the `Offset` variable is updated by adding the value of the `Length` field of the `PadN` option, plus 2 ([3]). If the `Length` field of the option is 0xFE, then 0x100 (0xFE + 2) will be added to `Offset`. But the result of that addition is truncated to an `UINT8` (the type of the `Offset` variable), which means that when adding 0x100 to `Offset` it will remain unmodified, and thus an infinite loop ensues, since execution can never break out of the `while` loop at [1] as seen below:

```
Ip6IsOptionValid (
  IN IP6_SERVICE  *IpSb,
  IN NET_BUF      *Packet,
  IN UINT8        *Option,
  IN UINT8        OptionLen,
  IN UINT32       Pointer
  )
{
```

```
    UINT8  Offset;
    [...]
[1]  while (Offset < OptionLen) {
        OptionType = *(Option + Offset);

        switch (OptionType) {
        [...]
[2]     case Ip6OptionPadN:
            //
            // It is a PadN option
            //
[3]         Offset = (UINT8)(Offset + *(Option + Offset + 1) + 2);
            break;
```

Bug 06 - edk2/NetworkPkg: Buffer overflow when processing DNS Servers option in a DHCPv6 Advertise message

Function `PxeBcHandleDhcp6Offer` in `NetworkPkg/UefiPxeBcDxe/PxeBcDhcp6.c` handles DHCPv6 offers made by DHCPv6 servers in response to EDK2 `Solicit` messages. The function allocates a pool buffer with the size given by the `OPTION_DNS_SERVERS` (0x17) option length at [1], which is controlled by the DHCPv6 server, then at [2] it calls `CopyMem` with a fixed size: `sizeof (EFI_IPv6_ADDRESS)`, which is 0x10. This means that if the length of the `OPTION_DNS_SERVERS` option included in the server response is shorter than 0x10, then that leads to a buffer overflow.

In NetworkPkg/UefiPxeBcDxe/PxeBcDhcp6.c:

```
PxeBcHandleDhcp6Offer (
  IN PXEBC_PRIVATE_DATA  *Private
  )
{
    [...]
    // First try to cache DNS server address if DHCP6 offer provides.
    //
    if (Cache6->OptList[PXEBC_DHCP6_IDX_DNS_SERVER] != NULL) {
[1]     Private->DnsServer = AllocateZeroPool (NTOHS (Cache6->OptList[PXEBC_DHCP6_IDX_DNS_SE
        if (Private->DnsServer == NULL) {
            return EFI_OUT_OF_RESOURCES;
        }

[2]     CopyMem (Private->DnsServer, Cache6->OptList[PXEBC_DHCP6_IDX_DNS_SERVER]->Data, size
  }
```

Bug 07 - edk2/NetworkPkg: Buffer overflow when handling Server ID option from a DHCPv6 proxy Advertise message

Function `PxeBcRequestBootService` in `NetworkPkg/UefiPxeBcDxe/PxeBcDhcp6.c` builds and sends a request to retrieve the boot file, and parses the reply.

In NetworkPkg/UefiPxeBcDxe/PxeBcDhcp6.c:

```
EFI_STATUS
PxeBcRequestBootService (
  IN  PXEBC_PRIVATE_DATA  *Private,
  IN  UINT32              Index
  )
{
[...]
    EFI_PXE_BASE_CODE_DHCPV6_PACKET  *Discover;
[...]

[1] Discover = AllocateZeroPool (sizeof (EFI_PXE_BASE_CODE_DHCPV6_PACKET));
    if (Discover == NULL) {
        return EFI_OUT_OF_RESOURCES;
    }

    //
    // Build the request packet by the cached request packet before.
    //
    Discover->TransactionId = IndexOffer->Dhcp6.Header.TransactionId;
    Discover->MessageType   = Request->Dhcp6.Header.MessageType;
    RequestOpt              = Request->Dhcp6.Option;
    DiscoverOpt             = Discover->DhcpOptions;
    DiscoverLen             = sizeof (EFI_DHCP6_HEADER);
    RequestLen              = DiscoverLen;

    //
    // Find Server ID Option from ProxyOffer.
    //
[2] if (Private->OfferBuffer[Index].Dhcp6.OfferType == PxeOfferTypeProxyBinl) {
[3]     Option = PxeBcDhcp6SeekOption (
                   IndexOffer->Dhcp6.Option,
                   IndexOffer->Length - 4,
                   DHCP6_OPT_SERVER_ID
                   );
        if (Option == NULL) {
          return EFI_NOT_FOUND;
        }

      //
```

```
      // Add Server ID Option.
      //
      OpLen = NTOHS (((EFI_DHCP6_PACKET_OPTION *)Option)->OpLen);
[4]   CopyMem (DiscoverOpt, Option, OpLen + 4);
      DiscoverOpt += (OpLen + 4);
      DiscoverLen += (OpLen + 4);
    }
```

At [1] the function allocates a buffer in the pool to hold an `EFI_PXE_BASE_CODE_DHCPV6_PACKET` structure, which is defined this way in `MdePkg/Include/Protocol/PxeBaseCode.h`:

```
///
/// DHCPV6 Packet structure.
///
typedef struct {
  UINT32    MessageType   : 8;
  UINT32    TransactionId : 24;
  UINT8     DhcpOptions[1024];
} EFI_PXE_BASE_CODE_DHCPV6_PACKET;
```

If the DHCPv6 offer it has received is a proxy one ([2]), then the function searches for the `SERVER_ID` option contained within said DHCPv6 offer ([3]), and proceeds to copy it to `Discover->DhcpOptions` (the buffer that was previously allocated in the pool at [1]), blindly trusting the option length as the size of the copy, which is fully controlled by the DHCPv6 server ([4]).

As a result, if the `SERVER_ID` option of the DHCPv6 proxy offer is longer than 1024 bytes (the size of the `DhcpOptions` member of the `EFI_PXE_BASE_CODE_DHCPV6_PACKET` struct), it results in a pool buffer overflow.

Bug 08 - edk2/NetworkPkg: Predictable TCP ISNs

Predictable TCP Initial Sequence Numbers (ISNs) are known to be a security issue since 1985 (see Morris1985 and Belovin1989) and a practical attack that exploits it was described in 1995 by Laurent Joncheray (Joncheray1995). A real world instance of an attack that exploited weak TCP ISNs to inject data in a TCP session to compromise a system was disclosed in 1995 (Shimomura1995).

In view of these and other findings RFC1948 proposed an algorithm to generate TCP ISNs in a way that prevents the described attack. In 2012 RFC6528 officially updated the TCP protocol specification and the new TCP ISN algorithm achieved status of Proposed Standard. In August 2022, RFC 9293 a revision of the core TCP specification was published as the new Internet Standard and aforemenetioned algorithm was recommended for the generation of TCP ISNs.

The TCP implementation in Tianocore's EDK2 IP stack generates trivially predictable TCP Initial Sequence Numbers and it is therefore vulnerable to TCP session hijack attacks.

The ISN for a new TCP instance is populated in `TcpInitTcbLocal` by calling `TcpGetIss()` as shown below in file TcpMisc.c

```
TcpInitTcbLocal (
  IN OUT TCP_CB  *Tcb
  )
{
  //
  // Compute the checksum of the fixed parts of pseudo header
  //
  if (Tcb->Sk->IpVersion == IP_VERSION_4) {
    Tcb->HeadSum = NetPseudoHeadChecksum (
                     Tcb->LocalEnd.Ip.Addr[0],
                     Tcb->RemoteEnd.Ip.Addr[0],
                     0x06,
                     0
                     );
  } else {
    Tcb->HeadSum = NetIp6PseudoHeadChecksum (
                     &Tcb->LocalEnd.Ip.v6,
                     &Tcb->RemoteEnd.Ip.v6,
                     0x06,
                     0
                     );
  }

> Tcb->Iss    = TcpGetIss (); /** new ISN is populated **/
  Tcb->SndUna = Tcb->Iss;
  Tcb->SndNxt = Tcb->Iss;

  Tcb->SndWl2 = Tcb->Iss;
  Tcb->SndWnd = 536;
```

Function `TcpGetIss` simply increments a global counter using a fixed increment as seen below:

```
TCP_SEQNO
TcpGetIss (
  VOID
  )
{
  mTcpGlobalIss += TCP_ISS_INCREMENT_1;
  return mTcpGlobalIss;
}
```

The global variable `mTcpGlobalIss` is initialized to a fixed value in line 23 of TcpMisc.c

```
TCP_SEQNO  mTcpGlobalIss = TCP_BASE_ISS;
```

The global variable is also updated with a fixed increment by the timer in TcpTimer.c

```
VOID
EFIAPI
TcpTickingDpc (
  IN VOID  *Context
  )
{
  LIST_ENTRY  *Entry;
  LIST_ENTRY  *Next;
  TCP_CB      *Tcb;
  INT16       Index;

  mTcpTick++;
  mTcpGlobalIss += TCP_ISS_INCREMENT_2;
```

The values `TCP_BASE_ISS`, `TCP_ISS_INCREMENT_1` and `TCP_ISS_INCREMENT_2` are defined in TcpMain.h:

```
///
/// The implementation selects the initial send sequence number and the unit to
/// be added when it is increased.
///
#define TCP_BASE_ISS         0x4d7e980b
#define TCP_ISS_INCREMENT_1  2048
#define TCP_ISS_INCREMENT_2  100
```

Therefore Tianocore's EDK2 TCP implementation generates ISNs using fixed increments from a fixed base value and thus is suceptible to TCP session injection and session hijack attacks.

Bug 09 - edk2/NetworkPkg: Use of a Weak PseudoRandom Number Generator

The EDK2 IP stack uses a pseudorandom number generator which is defined in `Network/Include/Library/NetLIb.h` as:

```
#define NET_RANDOM(Seed)  ((UINT32) ((UINT32) (Seed) * 1103515245UL + 12345) % 4294967295UL)
```

Throughout the NetworkPkg stack the above macro is used with `Seed` usually taking the value from the `NetRandomInitSeed()` function defined in NetworkPkg/Library/DxeNetLib/DxeNetLib.c as follows:

```
UINT32
EFIAPI
NetRandomInitSeed (
  VOID
```

```
  )
{
  EFI_TIME  Time;
  UINT32    Seed;
  UINT64    MonotonicCount;

  gRT->GetTime (&Time, NULL);
  Seed  = (Time.Hour << 24 | Time.Day << 16 | Time.Minute << 8 | Time.Second);
  Seed ^= Time.Nanosecond;
  Seed ^= Time.Year << 7;

  gBS->GetNextMonotonicCount (&MonotonicCount);
  Seed += (UINT32)MonotonicCount;

  return Seed;
}
```

The above function outputs an integer value based on the platform's clock time at which the function is called and the platform's monotonic counter. The C language idiom `NET_RANDOM (NetRandomInitSeed()` is used in several `NetworkPkg` functions to generate supposedly random unique numeric identifiers for various network protocol fields. Generating numbers in such a way does not produce random numbers with an uniform distribution, instead it produces easily predictable numbers and a biased distribution. Furthermore, the use of the same idiom in different network layers to assign allegedly random values to protocols fields or object make it possible for an external observer to recover the internal state of the generator by obtaining samples of the numbers used in any such network lawyer.

The idiom is used to generate DNS query IDs in `ConstructDNSQuery`:

```
ConstructDNSQuery (
  IN  DNS_INSTANCE  *Instance,
  IN  CHAR8         *QueryName,
  IN  UINT16        Type,
  IN  UINT16        Class,
  OUT NET_BUF       **Packet
  )
{
 ...
  //
  // Fill header
  //
  DnsHeader                   = (DNS_HEADER *)Frag.Bulk;
  DnsHeader->Identification   = (UINT16)NET_RANDOM (NetRandomInitSeed ());
  DnsHeader->Flags.Uint16     = 0x0000;
  DnsHeader->Flags.Bits.RD    = 1;
```

```
  DnsHeader->Flags.Bits.OpCode = DNS_FLAGS_OPCODE_STANDARD;
  DnsHeader->Flags.Bits.QR     = DNS_FLAGS_QR_QUERY;
  ...
```

To initialize the value of the IPv4 IP ID field in `Ip4DriverBindingStart()`

```
//
  // Initialize the IP4 ID
  //
  mIp4Id = (UINT16)NET_RANDOM (NetRandomInitSeed ());
```

which is the simply incremented for each outgoing IPv4 datagram in `Ip4Output()`

```
  // Before IPsec process, prepared the IP head.
  // If Ip4Output is transmitting RawData, don't update IPv4 header.
  //
  HeadLen = sizeof (IP4_HEAD) + ((OptLen + 3) & (~0x03));

  if ((IpInstance != NULL) && IpInstance->ConfigData.RawData) {
    RawData = TRUE;
  } else {
    Head->HeadLen = (UINT8)(HeadLen >> 2);
    Head->Id      = mIp4Id++;
    Head->Ver     = 4;
    RawData       = FALSE;
  }
```

The same idiom is used to initialize the fragment ID for IPv6 fragmentation Extension Headers, to generate DHCP transaction IDs on both Dhcp4 and Dhcp6 code, to obtain ephemeral port numbers for UDP and TCP, and in other parts of the stack. The security and privacy vulnerabilities arising from use of a weak pseudorandom numbre generator in various Internet Protocols are described in RFC 9414 and a number of algorithms to address those issues are suggested in RFC 9415. In this particular case, the use of a weak PRNG could facilitate DNS and DHCP poisoning attacks, information leakeage, denial of service and data insertion attack at the IPv4 and IPv6 layer (due also to the use of a per datagram unit increment of the corresponding ID fields).