EVASIVE INTERNET PROTOCOL: END TO END PERFORMANCE

By
Maaz Khan

Submitted in partial fulfillment of the requirements for the
Degree of Master of Science

Thesis Advisor: Prof. Michael Rabinovich

Department of Electrical Engineering and Computer Science
CASE WESTERN RESERVE UNIVERSITY

August 2011
We hereby approve the thesis/dissertation of

Maaz Khan

candidate for the Masters degree *

(signed) Michael Rabinovich
(chair of the committee)

Mehmet Koyuturk

Vincenzo Liberatore

(date) 06/09/2011

*We also certify that written approval has been obtained for any proprietary material contained therein.
Contents

Chapter 1  Introduction

1.1 Basic Approach ................................................................. 2
1.2 Security and feasibility ...................................................... 3

Chapter 2  Architecture ...................................................... 6

2.1 Delegation of Authority ..................................................... 6
2.2 T-Address ................................................................. 8
2.3 Datagrams ................................................................. 9
2.4 Obtaining a Destination T-address. ................................. 13

Chapter 3  Packet Modification using Netfilter Framework 15

3.1 The Packet Buffer .......................................................... 15
3.1.1 SKB basic management function .................................. 18
3.2 The Netfilter API ......................................................... 20
List of Figures

1. T-Address ................................................................. 9
2. Type-1 Datagram ...................................................... 11
3. Type-2 Datagram ...................................................... 11
4. Type-3 Datagram ...................................................... 12
5. Skbuff Boundaries ................................................... 17
6. SKB management functions ....................................... 19
7. Netfilter Hooks ....................................................... 22
8. EIP Prototype .......................................................... 30
9. Experiment Setup ..................................................... 33
10. Type -1 Overhead .................................................... 36
11. Transfer Rate TCP v/s EIP with router ......................... 37
12. Transfer Time TCP v/s EIP with router .......................... 37
13. Transfer Rate TCP v/s EIP without router .................... 38
14. Transfer Time TCP v/s EIP without router .................... 38
15. Transfer Rate TCP v/s EIP with router .......................... 40
16. Transfer Time TCP v/s EIP with router ......................... 41
17. Transfer Rate UDP v/s EIP without router .................... 41
18. Transfer Time UDP v/s EIP without router .................... 42
Evasive Internet Protocol: End to End Performance

Abstract

By

Maaz Khan

The current internet architecture allows hosts to send arbitrary IP packets across a network, which may not reflect valid source address information. IP spoofing and Denial of service attacks are ubiquitous. Filtering techniques are not sufficient enough to counter these attacks. Current Internet design calls for in-network authentication of addresses and attribution of traffic they generate.

Rabinovich and Spatscheck proposed a capability based architecture coined as Evasive Internet Protocol to overcome these issues by introducing transient addresses as an alternative to IP addresses to reach to a particular destination. In this architecture a destination can only be reached through a valid capability. The aim of this thesis is to implement Evasive Internet Protocol for the end hosts and measure the preliminary performance as compared to current internet protocols.
Chapter 1

Introduction

Today the Internet is assaulted from multiple fronts. Spam has already changed the social norms of using email, reflecting new assumption that legitimate mail might never be read by the recipient due to being entangled in spam filters. Malware dogs peer-to-peer networks and open source software distribution. Denial of service attacks against network infrastructures and Web sites have become routine. Computer break-ins and hijacking is wide-spread. Identity theft through phishing or break-ins is on the rise. The importance of these issues has been well recognized and is one of the main reasons behind the recent push by the networking community to re-engineer the Internet [1].

A fundamental challenge in designing a more secure Internet is to reconcile the security needs with preserving the openness of the Internet and privacy of its users. Indeed, preventing bad actors in an open environment seems to entail being able to hold actors accountable for their actions, which in turn suggests being able to attribute the actions to particular users, which undermines user privacy. Evasive Internet Architecture [2] aims to address this challenge, and to do so not by selecting a particular tradeoff point in this tussle space but by providing the tools that would give users the flexibility to select their own tradeoffs between openness and security, while at the same time not affect the privacy paradigm of the overall network architecture within which the approach is used (e.g., the current Internet).
1.1 Basic Approach

The approach relies on the general notion of capabilities [3, 4, 5]. In EIP, a capability is the only mean to reach a destination. The capability itself, which is an authorization to communicate with a host, is valid only for a specific sender and for a limited amount of time and data. The capabilities are distributed by a name system (e.g., DNS) which makes the design very effective as it uses the current name system with some feasible modification; this avoids extra infrastructure cost and maintenance.

Now since a host can only be reached through valid capabilities this calls for the source to replace the source IP address by its capability. Similarly, to reach a name server a source should have a valid capability which is issued by the higher naming servers in the DNS hierarchy. The capabilities become in effect the host addresses that are transient due to the limited validity of the capabilities. The name ‘Evasive Internet’ reflects the lack of permanent means of reaching a host.

This particular approach leaves the root name servers exposed for attacks since they need to be always reachable and thus demands a fixed capability to reach them. The authors of Evasive Internet Protocol [2] justify this vulnerability by leaving it to the root name server operators to handle these attacks which are quite justifiable because in current internet scenario the root name server is a frequent target and the name server operators have demonstrated that they are very much capable of handling these attacks.
1.2 Security and Feasibility

Given the tussle space between security, openness and privacy, the aim of EIP is to empower the internet end points to impose their own policies in this regards. Furthermore, the design strives for a minimal change in the current internet architecture that would allow this empowerment. The design itself is not a new overarching architecture for the internet; rather it can be used with the current internet architecture.

Although the architecture relies on capabilities to reach a host, the IP addresses would still be used by the existing routing protocols for forwarding table indexing and route computation. Thus routing protocols (example BGP) properties can still be retained and also the scalability properties related to topological information that is embedded in an IP address is also retained.

Although IP addresses are used for route computation and forwarding tables, the host IP address in EIP architecture cannot be used to communicate with the host. A compliant router will not forward a packet that has an invalid destination capability. This EIP address which in effect becomes a transient destination address is referred by the authors as T-address [2].

As far as privacy is concerned, EIP itself does not undermine it. Since EIP uses IP address to identify communicating parties, it can be said that the privacy of a user remains the same as it is in today’s internet.

Some of the security benefits that can be obtained by introducing EIP are:

- Currently anti-spoofing techniques rely mostly on ingress address filtering [6] but their effectiveness is reduced by concept like multihoming where a user can have an IP address from one ISP and uses another ISP connection to connect to the
internet, in such case ingress filtering often drops packet since the source IP for the packet and the network from which it is originating differs. Spoofing-based attacks continue to occur and exert damage. These attacks included old SYN-flood attacks and other DDoS attacks. Although specific mechanism have been proposed to counter some of these attacks, but the root of all these attacks, i.e. IP-forging, still exists. Now in the EIP approach the T-address of a destination is source-specific, thereby eliminating any attacks involving forged source address and thus making spoofing attacks organically impossible.

- The notion of capability enables recipients control over incoming flows because each host can implement fine-grained capability-issuing policies for particular external destinations. These policies can reflect various tradeoff decisions between security and openness. At the extreme, a host can only allow incoming traffic from a known set of destinations, and EIP will prevent other destinations from forging their IP addresses to bypass this policy. Short of this extreme, the recipient’s control allows a recipient to dynamically adjust the validity constraints granted to various external destinations based on their prior behavior. For example, this can stop continuous break-in attempts (e.g., password-guessing attacks) or denial of service attacks from a particular host.

- By enforcing a requirement that authoritative DNS servers only map host names to IP addresses with the authorization from the hosts’ network providers, Evasive Internet makes any fast-flux-hosted attacks impossible. In these attacks, the attacker uses compromised hosts as front-end Web servers. The attacker directs
clients to these Web servers by distributing a URL with the hostname that maps to the compromised hosts’ IP addresses, and using a public DNS service as authoritative DNS server maintaining these mappings. Typically, the front-end Web servers then collect phishing information from the users, or act as reverse Web proxies to a back-end fraudulent Web site in conducting fraudulent e-commerce transactions (so that the latter is hidden from detection). Much effort has been spent in trying to detect fast-flux networks \([7, 8, 9]\). Because of the difficulty of attribution, Clark and Landau includes them among the most dangerous attacks in today’s Internet \([10]\). Evasive Internet makes these attacks organically impossible.
Chapter 2

Architecture

The primary objectives of the design is (a) to prevent senders from forging their identities which is useful in itself by stopping spoofing attacks but more importantly it enables the next feature, (b) giving recipients full control over who can communicate with them and for how long and with what data constraints. While this architecture would not solve all the security issues of the internet by itself, but it would qualitatively improve internet security by eliminating wide range of attacks and would lay a very strong foundation for addressing other attacks at different levels.

The first iteration of this design was presented in a position paper [2]. The description below reflects the evolution of that design.

2.1 Delegation of Authority

Any entity issuing a destination’s t-address to another party must provide evidence that the issuer is authorized to issue the address. Otherwise, a rogue entity can issue a t-address to a destination without the destination’s consent, thus taking control from the destination over incoming traffic. Similarly, an entity requesting a t-address must provide proper credentials to the issuer, to allow the issuer to make a reasoned policy decision on granting the request, and also to grant the permission to the response to reach the requester.
To avoid any breach, EIP makes sure that whenever an entity obtains a block of IP address from a provider it also obtains a ‘slim’ certificate chain from the provider verifying the rights of the entity to use that block. The certificate chain emanates from ICANN which reflects the network provider hierarchy. Instead of the descriptive information present in a regular certificate, slim certificate is a small certificate estimated to be around 300 Bytes (256 Bytes for the signature and the rest for the IP block plus the hash of the recipient’s public key). Similarly an end host can obtain its certificate chain through DHCP.

There are two ways in which a source can obtain a T-address for a destination.

- **Name Servers** - name servers will be the primary entry points to the destinations whose names they maintain: these name servers will be issuing the initial t-addresses to these destinations in response to the DNS queries for the destinations. To this end, as part of the DNS service setup, for each block of IP addresses the name server will be handling, it must obtain a certificate from the corresponding ISP, which signs the IP address block. In most cases, the authoritative name server will belong to the same ISP as the destinations whose names it is resolving. In other cases, e.g., in content delivery networks such as Akamai [12], or when a domain name is handled by an external DNS service such as openDNS [13], the authoritative DNS server will map a name to destinations in different autonomous systems; in these cases, the name server would obtain a separate certificate for each address block from its corresponding provider.
• **End Host** – As discussed above when the host joins a network it can obtain certain rights through DHCP, with help of the rights obtained from DHCP the host can generate its own T-address. Now when a host initiates a connection with another host it includes its own source t-address in the EIP header so that when the destination receives that packet it can communicate back with the source via that issued source T-address. This property gives full control to the source on the constraints which the source wants to apply on the destination in the return flow. Thus in this way the Source host itself is issuing T-address to the destination host to communicate with the source.

### 2.2 T-Address

In this section we will discuss the main component of the architecture - the T-address, which is the only way to reach to a host. A T-address (Figure 1) contains the following fields:

- **Source IP Address** (16 bytes) – to incorporate IPv6 addresses.
- **Destination IP Address** (16 Bytes)
- **Validity Constraint** (12 Bytes) – A validity constraint consists of: 8 Bytes for time constraint and 4 Bytes for the number of bytes constraint. A time constraint is sufficient to carry a UNIX-like timestamp which can be used to timeout T-address with one second of granularity. A capability must satisfy both constraints to be valid.
• **Signature (256 Bytes)** – This field consists of a signature of the above three fields signed by the private key of the destination host or the authoritative name server.

• **Issuer’s Public Key (256 Bytes)** – To verify the signature.

• **Slim Certificate**

<table>
<thead>
<tr>
<th>16B</th>
<th>16B</th>
<th>12B</th>
<th>256B</th>
<th>256B</th>
<th>300 bytes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source IP addr</td>
<td>Dest IP addr</td>
<td>Validity constraint</td>
<td>(Fields 1-3) Signature</td>
<td>Issuer’s Pub. Key</td>
<td>Slim Certificate</td>
</tr>
</tbody>
</table>

**Figure 1. T-Address**

The design assumes that since there are very limited number of providers (a few thousand) so the providers public key and certificates are widely cached or can be obtained as needed using the hash as a key. This design leads to T-address of size 856 Bytes which leads to a datagram that are too large to be transmitted without fragmentation in today’s network which is of the order 1500 Bytes MTU. We will discuss in chapter 4 how we incorporated T-address in 1500 MTU.

### 2.3 Datagrams

Evasive Internet Protocol design calls for three types of datagram:

• **Type-1 Datagram (figure 2)** – This datagram forms the very first type of datagram that initiates connection between two hosts. It comprises of the following fields.
- **Type (1 Byte)**: Depending upon the type of the datagram it can have value 1, 2 or 3.

- **Upper protocol (1 Byte)**: This field is to indicate the kernel that which protocol the packet belongs to (e.g. 6 or 17 for TCP and UDP respectively).

- **Source T-address (856 Byte)**

- **Destination T-address (856 Byte)**

- **Delegation token (544 Byte)**: The use of this field will be discussed in section 2.5.

- **Flow ID (16 Byte)**: This is a 128 bit nonce (a number used only once with high probability) used to uniquely identify a particular communication in subsequent packets.

- **Flow TTL (8 Bytes)**: This is the TTL in the form of a UNIX-like timestamp used to timeout flow ID and rejects the datagram at the router which is processing that datagram.

- **Timestamp (8 Bytes)**: This is used for the originating time of the packet and is used to time-out TTL and discarding replayed packets with high probability, thereby thwarting the existence of man-in-the-middle attack.

- **Signature (256 bytes)**: This is the signature on the fields (Flow ID + TTL + timestamp) signed by the source private key, that allows the verification of the corresponding fields.
Figure 2. Type-1 datagram

The design assumes loosely synchronized clock among the hosts and the router. A host that is not properly configured may be penalized by this architecture design. Although this datagram is large in size (2546 Bytes) but since it is only used once to initiate a connection, its overhead doesn’t have a significant effect on the end to end data transfer performance. Its overhead is amortized by Type-2 datagram. In Chapter 4 we will discuss how we have implemented these datagrams within 1500 Bytes MTU.

- Type-2 Datagram (Figure 3) – It comprises of the following field:
  
  - Type (1Byte)
  
  - Upper Protocol (1 Byte)
  
  - Flow ID (16 Bytes)

Figure 3. Type-2 Datagram

This datagram is used for a period of Flow TTL for the same flow ID in subsequent communication. The objective of this datagram is to amortize the
overhead of Type 1 datagram. The preliminary assumption for the flow TTL is 5-10 seconds [2] and the majority of data transfer will be accompanied by these headers. A tradeoff exist in selecting the TTL time since keeping this TTL small will incur extra overhead frequently at the same time short life of these flow ID will be good choice for protection against fraudulent packets with same FID by man-in-middle attack. In our implementation we have limited Flow TTL to 10 seconds.

- **Type-3 Datagram (figure 4)** – Now since the sender knows the expiration time of a Flow ID since it assigns the TTL, therefore the sender must replace the current flow ID with a new flow ID before the current TTL expires to ensure uninterrupted communication, this replacement of flow ID is accomplished by Type-3 datagram.

<table>
<thead>
<tr>
<th>Type</th>
<th>Upper Protocol</th>
<th>Old Flow ID</th>
<th>New Flow ID</th>
<th>New Flow TTL</th>
<th>Timestamp</th>
<th>(New FID+TTL + timestamp) Signature</th>
<th>Payload</th>
</tr>
</thead>
<tbody>
<tr>
<td>1B</td>
<td>1B</td>
<td>16B</td>
<td>16B</td>
<td>8B</td>
<td>8B</td>
<td>256B</td>
<td></td>
</tr>
</tbody>
</table>

Figure 4. Type 3 Datagram

It comprises of the following field:

- **Type** (1 Byte)
- **Upper protocol** (1 Byte)
- **Old Flow ID** (16 Bytes)
- **New Flow ID** (16 Bytes)
- **New Flow TTL** (8 Bytes)
- 

12
- Time stamp (8 Bytes)

- Signature: This is a signature on fields (New FID + TTL +
  timestamp) in order to ensure the integrity because this information affects
  security for the entire next TTL period.

Type 3 datagram is 306 Bytes and it incurs acceptable computational
overhead since this verification of signature occurs once in every TTL
interval.

Finally, before a T-address is about to expire (because of the time constraint of T-
address), the sender may choose to include a subsequent-use T-address within type-2 or
type-3 datagram. This subsequent use will allow the source to have a continued running
communication with the destination without acquiring a new T-Address from the name
service thus making the distribution by name service application-oblivious and thereby
avoiding repeated name resolution overheads.

2.4 Obtaining a Destination T-Address

When a host needs to communicate with another host it must obtain that hosts t-address.
Since we discussed before that a source may obtain certificate chain via DHCP, through
which the host can generate its own t-address. Once the source has its own t-address then
it sends a DNS query to obtain the destination t-address to its Local Name server
(LDNS). The way LDNS works is that it routes the request in the name service hierarchy
on its own behalf (i.e. the LDNS behaves as a requester on behalf of the source and
provides its own t-address as the source of the query).
The catch here is that the reply to this query (i.e. destination t-address) will be destined to the LDNS IP address and not for the original source. In other words, if the source wants to use this t-address returned by the destination, it should provide certain rights that it can use this t-address which was actually meant for LDNS. This issue is been addressed by the use of delegation token field present in type-1 datagram.

This delegation token is provided by the LDNS to the client when it is returning non-client specific t-address, so that when a router receives this packet it may verify the authenticity of the source by checking this delegation token. The delegation token comprises of:

- Client and name server IP address (32 Bytes)
- Signature on both the IP address by the LDNS private key (256 Bytes)
- LDNS public key for verification (256 Bytes)
Chapter 3

Packet Modification using Netfilter Framework

This chapter will cover a background study on how Linux kernel handles packet and we will explore the ways how a packet can be altered on the fly within the kernel by some pre-well defined hooks residing inside the Linux kernel.

3.1 The Packet Buffer

The network implementation of Linux is designed to be independent of a specific protocol [16]. This applies both to the network adapter protocols (Token rings, Ethernet) and network and transport layer protocols (TCP/IP). Socket Buffers or skbuffs are the buffers in which the Linux kernel handles network packets. The packet is received by the network card, put into a skbuff and then passed to the network stack, which uses the skbuff all the time. This common data structure (struct sk_buff) is used by all the buffers and the network-related queues in the kernel and is probably the most important data structure in the networking code of Linux kernel, the data that has been transmitted or is about to be received is represented by the struct sk_buff header which is defined in the <include/linux/skbuff.h> include file.

The skbuff buffers are organized in the form of a doubly linked list, in such a way that makes it very efficient in moving a skb buffer element from the beginning or end of a list, or to the beginning or end of some other list. A queue is defined by struct sk_buff_head, which includes a head and a tail pointer to sk_buff elements. There are certain functions
that manage these queues like skb_queue_tail(), skb_queue_head() to add a skb buffer or skb_dequeue_tail(), skb_dequeue() that tends to operate on an sk_buff_head to remove a skb buffer.[17]

Whenever a new skb buffer is created, it gets its memory allocated from the kernel memory, which can be done by using functions like alloc_skb() or dev_alloc_skb(), where dev_alloc_skb() function is used only by the drivers. Similarly, the allocated buffer can be freed by calling kfree_skb(), this kfree_skb() is both called and invoked directly through the use of dev_kfree_skb wrapper. Dev_kfree_skb is defined for use by device driver that parallels dev_alloc_skb in name but consists of a simple macro that does nothing but call kfree_skb(). The basic functionality of this function is to release a buffer only when the skb->users counter is 1 (when no other user is using the buffer), Otherwise, the function simply decrements that counter. So if a buffer had two users, only the second call to dev_kfree_skb or kfree_skb would free the memory. [19]

Internally the sk_buff provides an additional management layer. The data space is divided into a head area and a data area. The head area is only for the headers (TCP, UDP, IP etc) which allow kernel functions to reserve space for the header so that the data doesn't need to be copied around. Therefore, after allocating an sk_buff, the header space is reserved by using skb_reserve().

When an application passes data to a socket, the socket internally creates an appropriate socket buffer to store the payload data in the variables of this structure. During its travel across the various layers within the kernel the packet headers of each layer are inserted in front of it. To make the insertion operation easier sufficient space is reserved for packet headers. The payload in the Linux kernel is copied only twice: once when the packet data
is passed to the network adapter, and once when it transits from the user space to the kernel space [16]. In Linux the free space in front of a packet data is called headroom, and the space behind the current packet data is called tailroom.

![Figure 5 skbuff boundaries](image)

The Boundaries in a skb is represented by 4 pointers (Figure 5):

- `Unsigned char * head;`
- `Unsigned char * end;`
- `Unsigned char * data;`
- `Unsigned char * tail;`

These above 4 pointers represent the boundaries of the buffer and the data within it. When each layer access and modify the buffer for its activities, it may allocate extra
space for a header or for data accordingly. head and end points to the beginning and end of the space allocated to the buffer respectively, while the beginning and end of actual data is pointed by data and tail respectively. The layer then further can fill in the gap between head and data with its specific protocol header, or the gap between tail and end with new data.

Apart from the above basic pointers the struct sk_buff has fields to point to the specific network layer headers:

- transport_header – for the transport layer which can include TCP, UDP, ICMP header and more
- network_header – for the network layer which can include IP, IPv6 or ARP header and more.
- mac_header – for the link layer headers

The functions skb_transport_header(skb), skb_network_header(skb), and skb_mac_header(skb) returns pointer to the corresponding protocol header.

3.1.1 Skb Basic Management Functions

We already seen in the above section how Linux kernel allocates memory for new buffers and how it further free the memory, also we saw how the header space is being reserved by using skb_reserve function. We will now see some basic and probably the most important functions (Figure.6) to manipulate the boundaries maintained by the 4 pointers mentioned in the previous section.

- skb_put (skb, len) – is a function that appends data to the end of the current data of a packet. More specifically, skb_put() increments the pointer tail
by length `len`. It is up to the caller to copy the correct data to the packet data space, `skb_put()` merely sets the pointers to a new position. Before calling `skb_put()`, we should confirm that the tailroom is sufficient otherwise the tailroom should be expanded by using `pskb_copy_expand()` function.

- `skb_push(skb, len)` - is similar to `skb_put()`, but increases the current packet data space at the beginning of the packet by `len` bytes. In other words, the data pointer is decremented by `len`, and `skb->len` is incremented by `len`. The return value of `skb_push()` points to the new data space. Again, the headroom size should be check first and before using this function.

```
Original skb

<table>
<thead>
<tr>
<th>Head</th>
<th>Data</th>
<th>Tail</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
After skb_push()

<table>
<thead>
<tr>
<th>Head</th>
<th>Data</th>
<th>Tail</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
After skb_pull()

<table>
<thead>
<tr>
<th>Head</th>
<th>Data</th>
<th>Tail</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
After skb_put()

<table>
<thead>
<tr>
<th>Head</th>
<th>Data</th>
<th>Tail</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

```
After skb_trim()

<table>
<thead>
<tr>
<th>Head</th>
<th>Data</th>
<th>Tail</th>
<th>End</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 6. skb management functions

- `skb_pull(skb, len)` - is used to truncate `len` bytes at the beginning of a packet. The pointer `skb->data` is adjusted, and the length of the packet `skb-
>len) is reduced accordingly but, first, we need check on whether there are still len bytes in the free part of the packet data space.

- **skb_trim (skb, len)** – is used to set the current packet data space to len bytes, which means that this space now extends from the initial occupancy of data to tail - data + len. Usually this function is called when we need to truncate data at the end.

Apart from these basic functions we have skb_tailroom and skb_headroom to obtain the tail and head room, and if needed the head and the tail room can be expanded by using skb_copy_expand function which we have extensively used during the implementation of Evasive Internet Protocol.

3.2 The Netfilter API

Netfilter is a subsystem that was introduced in the Linux 2.3 kernel. Netfilter makes network tricks such as packet filtering, network address translation (NAT) and connection tracking possible through the use of various hooks in the kernel's network code. These hooks are places where the kernel can register functions to be called for specific network events either by statically building it or in the form of a loadable module.

Netfilter provides a framework for packet mangling, outside the normal Berkeley socket interface. It’s the engine behind iptables the popular firewall solution for Linux. Firstly, each protocol defines hooks which are well-defined points in a packet’s traversal of that protocol stack. At each of these hooks or points, the protocol will call the netfilter
registered function with the packet and the hook number. Now, parts of the kernel can register to listen to the different hooks for each protocol. So when a packet is passed to the netfilter framework, it checks to see if anyone has registered for that protocol and hook; if so, they each get a chance to examine (and possibly alter) the packet in order, then discard the packet or allow it to pass.

3.2.1 Netfilter Hooks

Now since the netfilter modules can be loaded into the Linux kernel during runtime, it needs certain points or Hooks in between the actual packet route codes in order to enable dynamic hooking of the functions. The netfilter framework provides with certain hooks [18] for each supported protocol which is defined in the protocol-specific header file. The following five hooks are defined for IPv4 in <linux/netfilter_ipv4.h>: (Figure.7)

- **NF_IP_PRE_ROUTING** (default value is 0): incoming packets pass this hook in the ip_rcv() (linux/net/ipv4/ip_input.c) function after the sanity check but before they are processed by the routing code.

- **NF_IP_LOCAL_IN** (default value is 1): after the routing code if the incoming packet is addressed to the local computer than it pass this hook in the function ip_local_deliver().

- **NF_IP_FORWARD** (default value is 2): all incoming packets not addressed to the local computer pass this hook in the function ip_forward().

- **NF_IP_LOCAL_OUT** (default value is 3): all outgoing packets that were created by the local processes pass this hook in the function ip_build_and_send_pkt().
- **NF_IP_POST_ROUTING** (default value is 4): this hook in `ip_finish_output()` function represents the last chance to access all outgoing packets just before they hit the wire.

```
  |          ^
  |        [ROUTE]
  v        |
[2] [5]
  |          ^
  |        [INPUT*] [OUTPUT*]
  v        |
[3] [4] [5] [2][1][4]
```

Figure.7 Netfilter Hooks

Also calling the **NF_HOOK** macro causes the routing code to process the filter functions hooked into a netfilter hook. More specifically, the **NF_HOOK** macro has the following arguments:

- **pf** (*protocol family*): This is to identify the protocol family `PF_INET` for IP Version 4, `PF_INET6` for IP Version 6.
- **hook**: This is to identify the hook. All the valid identifiers are defined in a header file `linux/netfilter_ipv4.h`. 
• skb: This represent the pointer to the sk_buff structure with the packet to be handled.

• indev (input device): This is a pointer to the net_device structure for the network device that is responsible for receiving the packet.

• outdev (output device): This is a pointer to the net_device structure for the network device that is responsible for sending the packets.

• okfn(): This function gets invoked when the filter functions registered with this hook returned NF_ACCEPT, thereby okaying the packet’s transit.

The NF_IP_LOCAL_IN and the NF_IP_LOCAL_OUT hooks are used in the implementation of Evasive Internet protocol, eip_in and eip_out module are the our kernel modules that uses these 2 hooks respectively in order to append and strip of EIP headers from the packet. In the next chapter will be covering these modules in detail.

3.2.2 Registering and Unregistering Hook Functions

The packet-filter functions that are actually hooked into the netfilter hooks are so-called hook functions [18], are of the type nf_hookfn. The signature of which is defined in <linux/netfilter.h> as follows:
The packet-filter function returns a value that specifies what should happen to the packet. It is of the type unsigned int and can take any of the following values, defined in `<linux/netfilter.h>`:

- **NF_DROP** (default value is 0): This is used to inform the function to drop stop the active rules list and drop the packet.
- **NF_ACCEPT** (default value is 1): This is used to accept a particular packet and pass it to the next packet filter function in the rules list. The packet is released by `okfn()` for further processing, once the end of the list has been reached.
- **NF_STOLEN** (default value is 2): In contrast to NF_DROP, this is used to inform the kernel that it should forget about the packet and the hook function withholds it for later processing.
- **NF_QUEUE** (default value is 3): This queues the incoming or the outgoing packets in a queue with the help of `nf_queue()` function and then can be removed and processed at a later time. `nf_reinject()` is invoked to return the packet to the Linux kernel.
• NF_REPEAT (default value is 4): In contrast to NF_ACCEPT, this is used to repeat the processing of the packet in the hook function.

Now registering and unregistering a packet-filter function within the Linux kernel is done by `nf_register_hook()`, `nf_unregister_hook()` functions. The parameter passed is an `nf_hook_ops` structure, which includes all information required.

To register a new packet-filter function with the Linux kernel, we first have to initialize a structure of the type `nf_hook_ops` (linux/netfilter.h) with all of the management information required:

```c
struct nf_hook_ops
{
    struct list_head list;
    nf_hookfn *hook;
    int pf;
    int hooknum;
    int priority;
};
```

The fields of this structure `nf_hook_ops` have the following meaning:

- list: This is a linked list which is used to handle the `nf_hook_ops` structures within the Linux kernel
- hook(): this is a pointer to the packet-filter function.
- pf: the protocol family identifier (e.g., PF_INET or PF_INET6)
• **hooknum**: This is the hook identifier (e.g., `NF_IP_INPUT`) which is used to determine what place or hook the packet-filter function should listen.

• **priority**: This specifies the priority of the packet-filter functions within the rules list of a hook.

Now that we have seen how the packet is handled by the Linux kernel internally through skb buffers and how a packet can be captured and altered at a certain point inside the kernel, in the next chapter we will go through how the Evasive Internet protocol has been implemented using the above discussed methods.
Chapter 4

Implementation

This chapter describes the implementation details for the Evasive Internet protocol. The Evasive Internet Protocol itself can be divided into three major subcomponents that can be largely decoupled.

- The first subcomponent is the modified DNS, which is responsible for resolving and distributing transient addresses.
- The second component is the implementation of EIP on individual hosts, which includes mechanisms for continuing communication as transient addresses expire.
- The last component is the routing of EIP packets, which involves checking transient addresses for validity and dropping packets that fail the check.

The focus of this thesis is on the second component which is to develop a prototype of EIP for the End host communication and perform preliminary performance compared to TCP and UDP protocols.

As described earlier type-1 datagram are approximately of the size 2.5 Kbytes. This size is too large to be transmitted without fragmentation since the usual MTU is 1500 Bytes. For the purpose of our prototype and to test a more complete implementation we used 128 Bytes key encryption instead of 256 Bytes encryption. While the world is moving to jumbo frames as standard, 128B keys are strong in the interim. This change leaves us with type-1 datagram of the size 1394 Bytes which is under the 1500 Bytes MTU.
The DNS component for EIP has not been implemented, as it itself calls for a separate research project. For the purpose of our End to End host prototype, we have used a Dummy DNS which has pre-build source and destination t-address. To perform the signature in the pre-build t-address, OpenSSL command line tool [14] with 128 Bytes key encryption is used.

4.1 Components

We will now discuss the components in the EIP prototype (Figure.8):

- EIP Application level Module – This is the application level program that initiates/accepts the initial TCP/UDP connection. EIP type-1 and type-3 headers call for RSA encryption at the kernel level. Unfortunately, the Linux kernel Crypto API does not provide RSA encryption at the kernel space, thereby not allowing the EIP kernel module to sign the fields in type-1 and type-3 datagrams. To bridge this gap we generate the EIP headers within the EIP application module component by calling a function EIP_Socket() and then pass headers to the kernel through a memory mapped file. The purpose of EIP_Socket() is to sign the fields in the type-1 and type-3 headers and then save them in EIP mapped file. EIP_Socket() internally calls a function create_taddress_headers() which signs and generates EIP headers and then initiate a normal socket connection. In the actual implementation for EIP we envision that the EIP_Socket obtains the t-address along with the delegation token from the DNS and then passes them to the kernel socket structure from where they are further made available to the lower network layers. We used OpenSSL API [14] for signing the fields. Apart from
this, EIP application component is also responsible for changing the Maximum Segment Size (MSS) to 1300 for the ongoing communication. The purpose of this MSS change will be discussed in section 4.2.

- EIP mapped File – Due to the limitation of headers been signed at the application layer we need some mechanism to transfer these headers from application layer to the kernel space so that we can hand over these headers to the EIP kernel module. To accomplish this task, EIP application saves the generated headers from create_taddress_headers() into a file which is further used as a memory mapped file “EIP mapped file” by the EIP kernel module. Thus in this way we make sure that the signed EIP headers are available at kernel space. The reason a memory mapped file is used is because we want all the headers to be inside the main memory that are present in the file so that obtaining data from the file will be like obtaining data from the main memory instead of performing frequent I/O operations. As we mentioned in the previous component that in real world EIP implementation we envision that the t-address can be passed to the lower layers via kernel socket structure instead of using memory mapped file.

- EIP kernel Module (EKM) – This is the most important component of the EIP prototype. The primary objective of this module is to add EIP header when a packet leaves host machine and to remove the EIP headers when a packet is received by a host machine and then forward them to the upper level in the kernel. To accomplish this task Netfilter framework is been used. We discussed in previous chapter how netfilter inserts hooks within the packet flow in the kernel.
When a packet leaves a host machine it is been captured just before it is about to hit the wire by using NF_IP_POST_ROUTING hook. The captured packet is an SKB buffer which consists of certain boundaries and areas specific to header and data. Now before a modification is done on the packet the EIP kernel module makes sure that:

- The buffer is not paged in the memory; it is a common practice by the Linux kernel to page a buffer instead of storing it in a contiguous space. If the buffer is been paged than the EIP kernel module linearize it to contiguous space so that further modification on the packet can be done.

![Figure 8: EIP prototype](image)
- Once the linearization on the buffer is done, the EIP Kernel Module expands the buffer with size type-1, 2, or 3 depending upon the packet. If it is a SYN packet it indicates that it is the start of the connection thus the kernel module expands the buffer with size equal to type 1 datagram or else the module expands it for type-2 datagram, also simultaneously the module checks the current time against and Flow TTL and makes sure that it is within the bounds and when the TTL expires the kernel module expands the buffer to accommodate type-3 datagram to renew the Flow ID.

- The EIP kernel module generates a hash table using the information inside the EIP Mapped file. The hash table is generated using uthash [15] library, and the table maps type-1, 2, 3 headers and the Flow TTL corresponding to each destination IP addresses that are available in the EIP mapped file. Thus when the kernel module captures a packet it uses the destination IP address of the packet as a key to obtain corresponding EIP header information from the hash table and then add the appropriate header at the end of the packet and then returns NF_ACCEPT to the hook function signaling the hook that the packet is now ready to be transmitted over the wire. Although we envision that in real world the EIP headers would be in between Transport and Network layer, the reason why we are adding the EIP headers at the end of a packet in our prototype is because we don’t want to incur extra overhead by moving the IP payload data for every packet on the fly and then insert EIP header in between.
Similarly the EIP kernel module uses NF_IP_PRE_ROUTING hook to capture a packet sent by the host from the other end and then strip of the EIP header from it and return NF_ACCEPT to the hook function signaling it to forward the packet to the upper layers in the kernel.

4.2 Changing the MSS

As mention in the previous section, one of the functionalities of the EIP application is that it changes the MSS to 1300 Bytes for the ongoing communication. The reason why this change is done is because since the MTU is 1500 Bytes and say if we want to add the Type-3 header which is 178 Bytes at the end of a packet it will result in a packet size of 1678 Bytes which is out of the MTU range and thus the packet will not leave the interface. So in order to avoid this drop the MSS is changed to 1300 Bytes so that when type-3 header is added to it, it will still be in the 1500 Bytes MTU range.

A possible question that arises here is: what will happen when type-1 header which is 1394 Bytes is added to a packet. In our prototype we have used t-address that has sufficient validity constraint to last throughout the file transfer, thus type-1 headers are added only to SYN packets which are usually of the size around 70 Bytes, so when we add a type-1 header to it, it results in packet size of around 1470 Bytes which is still under the 1500 Bytes MTU limitation. In the real world EIP implementation type-1 headers does not necessary apply only to SYN packets, they will be needed again, once the validity constraint of a t-address is exhausted.
This concludes our chapter on Implementation, in the next chapter we will see the preliminary performance of this prototype as compared to TCP and UDP file transfer rate and transfer duration.
Chapter 5

Performance

To evaluate the performance of EIP prototype we used two metrics. The first metric is based on the transfer rate and the second metric is based on the transfer time. We compare the EIP performance for these metrics with the TCP and UDP performance.

5.1 Experiment setup

Figure.9 Experiment Setup (a) without router, (b) with router

Figure.9 (a) shows the test setup network without router and Figure.9 (b) shows the test setup with router. The test involves UDP performance which does not guarantee delivery of packets over large networks. Packets drop and network congestion may incur extra delays, also the prototype demands for modification in the router processing. For all these reasons the test bed is chosen as an isolated segment setup on a local LAN. The router used for datagram processing is Dell Power Edge server with a 2.4GHz Intel Xeon E5620 processor, 8GB RAM, with Gigabit Ethernet NIC running Linux 2.6 Kernel. The two end
hosts used for data transfer are Intel Core2 Duo T6500 @ 2.10 Ghz, 4 GB RAM and Dell latitude Intel Pentium M processor, 1 GB RAM both running Linux 2.6 Kernel. File transfer is performed over 100Mbps wire speed. The modification of the router and the processing of EIP packets at the router is implemented by Eamon Johnson in a separate parallel project.

For the purpose of performance we have used file size ranging from 1 Mbytes to 128 Mbytes for UDP, TCP and EIP rate and time measurements.

5.2 TCP v/s EIP performance

Figure.11 shows EIP transfer rate performance as compared to TCP performance measured at router. The worst performance obtained with EIP was 58 Mbps, while the worst performance in the case of legacy was 70Mbps. The results look very promising and feasible to be applied in current systems.

Figure.12 shows EIP transfer time performance as compared to TCP transfer time measured at the router. Similarly Figure.14 displays the same measurement without router. In the case where router was included, it resulted in an average increase of 5% in time for file transfer, whereas a 3% increase was obtained in the case where router was not involved. This increase is acceptable given the extra layer of security provided by EIP.

The EIP versus TCP transfer rate performance without the router is displayed in Figure.13. The worst performance reported with EIP was 69Mbps as compared to legacy case which was 80Mbps. The worst EIP performance in this case is better than the worst performance of EIP with router, and it should be because of the extra processing done at
the router. Interestingly the rate decreases with the increase in file size; we are not sure what exactly the cause for this is, and required further investigation to understand this behavior.

In order to measure the latency, we considered type-1 datagram, which was the worst case scenario. To accomplish this measurement we measured the TCP handshake time in both the cases, that is with and without router and found that the increase in the time is quite acceptable. In the case where the router was involved, there was a 1ms second delay in the TCP handshake communication, whereas without the router the delay was 0.35 ms. The delay in the former case was due to signing the data at the end host and verifying the signature at the router, whereas the delay in the latter was due to signing the data at the end host. As we mentioned earlier that type-1 datagram are not so frequent during the data transfer, most of the communication is carried via type-2 datagram. Thus type-1 datagram won’t have any significant effect overall during a file transfer. Below is the time handshake packet leaves and arrives at a host (in seconds):

<table>
<thead>
<tr>
<th></th>
<th>EIP with router</th>
<th>Legacy with router</th>
<th>EIP without router</th>
<th>Legacy without router</th>
</tr>
</thead>
<tbody>
<tr>
<td>SYN</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>SYN, ACK</td>
<td>0.001402</td>
<td>0.000479</td>
<td>0.000494</td>
<td>0.000168</td>
</tr>
<tr>
<td>ACK</td>
<td>0.001449</td>
<td>0.000501</td>
<td>0.000546</td>
<td>0.000190</td>
</tr>
</tbody>
</table>

Figure.10 Type-1 Overhead
Figure 11: Transfer Rate TCP v/s EIP with router

Figure 12: Transfer Time TCP v/s EIP with router
Figure 13 Transfer Rate TCP vs EIP without router

Figure 14 Transfer Time TCP vs EIP without router
5.3 UDP v/s EIP performance

For the UDP v/s EIP test we used the same files to transfer as TCP v/s EIP, with the only difference that we limited the outgoing rate for the host to 50 Mbps. The reason for this limitation was because there was packet loss if the data was sent above 50 Mbps rate from the host. In the case of TCP its internal flow control mechanism was enough to handle the sending rate but since UDP doesn’t have any such mechanism we limited the outgoing traffic rate from the host to 50 Mbps by using the following command at the host interface:

```
# Tc qdisc add dev eth0 root tbf rate 50 mbit burst 10 kb latency 1s
```

Figure.15 shows EIP transfer rate performance as compared to UDP performance measured at the router and Figure.17 displays the same without the router been involved. The worst performance including router for EIP was 45.5Mbps and for legacy it was 46Mbps, and in the case where router was not involved it resulted in worst performance of 45.2Mbps for EIP and for legacy it was 46.2Mbps.

With the increase in file size, the transfer rate decrease for both EIP and UDP. We believe that the reason why the rate drop in UDP performance with increase in file size is because, we limited the rate to 50Mbps, now the way this limit is been accomplished by the Linux kernel is that it internally provides a queuing mechanism for traffic control over the interface. Packets leaving host are queued and then sent with the desired rate through the interface, now the more the file size is, the more huge this queue becomes. Thus more maintenance is required for handling the queue which results in the decrease in transfer rate.
Figure.16 shows EIP transfer time performance as compared to UDP performance measured at the router. An average increase of 4.8% in the time for file transfer was obtained when router was included. Figure.18 shows EIP transfer time performance as compared to UDP performance without the router. An average increase of 4.5% in the time for file transfer was obtained without router. This increase in the transfer time is less as compared to the increase in the transfer time as compared to the EIP versus TCP performance, which was due to the presence of acknowledgment packets in the TCP connection.

Figure.15 Transfer Rate UDP v/s EIP with router
Figure.16 Transfer Time UDP v/s EIP with router

Figure.17 Transfer Rate UDP v/s EIP without router
Figure 18: Transfer Time UDP vs EIP without router
Chapter 6

Conclusion

In this research project we presented the architecture, prototype and the preliminary evaluation of EIP for the end to end host. Although the DNS component of the EIP architecture is yet to be implemented and tested but the current results demonstrate a very promising future for EIP.

The current prototype is developed with certain limitation of signing the fields at the application layer and then sending them at the kernel space. This limitation will be hopefully overcome soon in near future as the Linux kernel developers have hinted to provide the RSA implementation at the kernel level in Crypto API.

Also hardware-implemented signature generators are available and definitely their cost will be reducing in near future, which can be used to sign each type-2 datagram thereby eliminating the concept of Flow TTL which was introduce to reduce the frequency of signature generation by the sender.
References


[10] David D. Clark and Susan Landau. The problem isn’t attribution; it’s multi-stage attacks. In The 3d Workshop on Re-Architecting the Internet (ReArch), 2010.


