# Global Information Assurance Certification Paper

# Assumptions in Intrusion Analysis
## Gap Analysis

*GCIA Gold Certification*

Author: Rodney Caudle, Rodney_Caudle@yahoo.com

Adviser: Joey Niem

Accepted: TBD

Outline

## 1. Abstract

This paper examines one of the assumptions that form the foundations of packet analysis. A discussion of an approach to analyzing protocol stacks is presented. This approach can be used to determine gaps in the protocol stack where an analyst can be misled. Through the discussion a gap in the TCP/IP protocol stack is examined revealing one of the common assumptions made in intrusion analysis; trusting the content of the protocol field of the IP header. An analysis and demonstration of exploiting this gap is presented in the main body of the paper and the source code is provided in the appendices. The paper concludes with information on mitigating the exposure presented through the analysis presented in the paper and relates the general approach used in the exploit to previous attempts captured by the Honeynet Project.

## 2. Introduction

Every decision that an analyst makes is built upon the interpretation of data. The data is interpreted based upon a foundation of assumptions. How does the analyst know what pieces of information are accurate and what pieces of information are false when analyzing the information collected? This paper examines one of the common assumptions made as an intrusion analyst looking at network packet captures and explores the possible avenues which could determine that the assumption may not be as trustworthy as has been previously assumed. This paper attempts to guide the analyst by providing a detailed analysis of the TCP/IP standards stack with particular focus on the communication that exists between layers of the stack. As will be shown in this paper, the communication, or lack of communication, provide the possibility of exploitation at various levels as data passes between layers in the standards stack. Understanding the potential exploitation points will assist the analyst to understand which pieces of information can be trusted when performing an analysis and which pieces should be considered suspect.

### *Terminology*

This paper uses the terms “standards stack” and “protocol stack” [13] interchangeably. A protocol is an implementation of a standard. Different implementations of the same

standard can lead to gaps which also can be exploitable. However this discussion is beyond the scope of this paper.

### *TCP/IP Model vs. OSI Model*

The Open Systems Interconnection (OSI) Model [1] is considered to be an ideal scenario for network communication where communication between two entities passes through a series of layers. Each layer has a set of rules, or protocol, which provide instructions on how to encode and decode data passing through the layer. On the sending end of the communication, each layer takes the information from the previous layer, then applies the instructions of encoding the data and passes it to the underlying layer. On the receiving end, each layer interprets the data according to the instructions in the protocol and the remainder is passed back up the stack. Communication begins at the application layer, travels down the stack of the sending entity to the physical layer. The communication is then broadcast using the physical layer to the receiving entity which reads the information by moving up the stack as far as needed. Depending on the role of the entity interpreting the current communication only part of the layers may need to be interpreted. For example, a router tasked with delivering packets to their destination might only interpret the first three layers (physical, data link and network) before making a decision. Figure 1 depicts the communication between two entities using the ideal OSI model interpreting all of the layers.

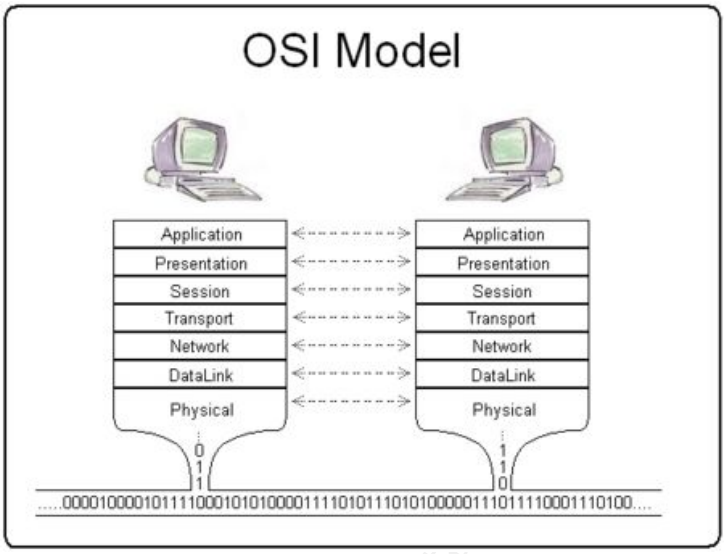


*Figure 1: OSI Communication Model [2]*

In contrast to the OSI model, TCP/IP model handles communication with fewer layers combining those layers which can only be separated in an ideal situation. For instance, the TCP/IP model joins the physical and data link layers of the OSI model into the Network Interface layer. This simplifies the delivery of packets on a TCP/IP network because the delivery of the packets on a physical network often drives the size of the frames built at the data link layer. Figure 2 shows the relationship between the TCP/IP model and the OSI model including the discrepancies between the two approaches.

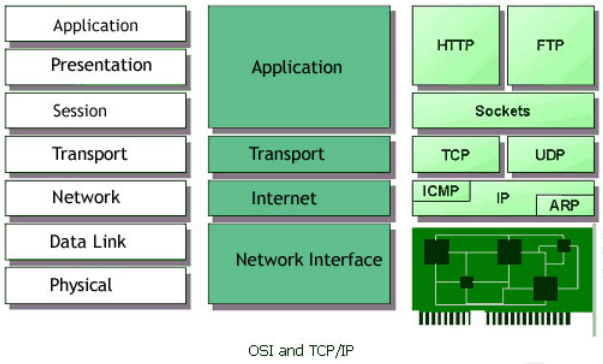


*Figure 2: TCP/IP Mode[3]*

As seen in figure 2, the TCP/IP model (center) to communication has fewer layers than the OSI Model (far left). However, TCP/IP has a myriad of protocols available with multiple possibilities for each layer (far right). This paper focuses on the communication between the layers of the TCP/IP model and will only relate the information to the OSI model where the two approaches overlap. For the purposes of this paper, the fewer the layers, the fewer chances of exploitation so the TCP/IP model represents fewer potential exploitation vectors than the OSI model.

### *Stackable Standards*

The idea of using a series of standards to form a communication or protocol stack is not a new concept. To fully comprehend the complex interactions of the various layers an analyst must first determine the series of standards which are being used, determine the limitations of each, and define the rules of interaction between the layers. For each layer of

the protocol stack there is a standard. If there are four layers to the stack there are, at least, four underlying standards. In its most basic form, the standard defines a transformation of data as it passes through the layer. In the case of TCP/IP there are more standards, especially for the network interface layer because multiple physical mediums have been built which support TCP/IP. This paper focuses on the four layers of communication for the TCP/IP standards stack: Network Interface, Internet, Transport and Application.

The Network Interface Layer, Layer 0, is the lowest layer of the TCP/IP protocol stack and for the purposes of this paper only the Ethernet standard (IEEE 802.3 [4]) and its use as a medium for the propagation of IP version 4 (RFC 894 [5]) will be covered. The next layer above is the Network Layer, Layer 1. For Layer 1, this paper focuses on the IP standard (RFC 791 [6]) and its capabilities for directing traffic between entities. The Transport Layer, Layer 2, has many standards available. This paper covers both UDP (RFC 768 [7]) and TCP (RFC 793 [8]) standards for communication. The final layer, Layer 3, called the Application layer covers HTTP (RFC 2616 [9]) and SMTP (RFC 821 [10], RFC 822 [11]) standards. Figure 2 (above) provides a nice overview of the standards (far right) next to both the TCP/IP model (center) and the OSI model (far left). Figure 2 will be used as a reference throughout this paper.

As data passes through a protocol stack, like the OSI Model or TCP/IP, the data

makes a series of progressions. At the highest layer, Layer 3, the data packet can be considered as a raw data packet. As the data progresses down the protocol stack toward the physical medium additional information is added at each consecutive layer. This additional information, called a header, provides both instructions for the layer the data is currently passing through as well as fulfilling any rules of interaction necessary to facilitate communication between layers.

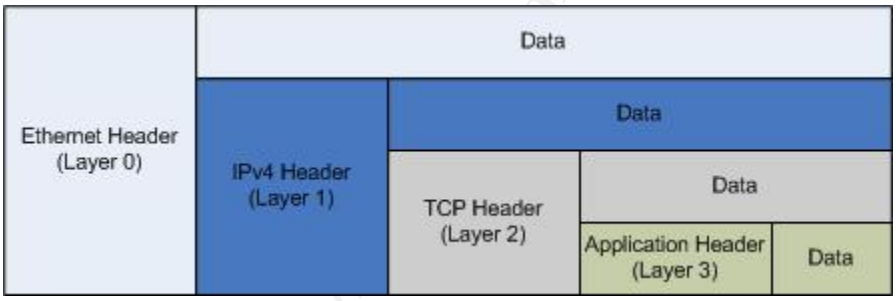


*Figure 3: TCP/IP Standard Stack*

Figure 3 shows the complete stack of standards in an example as it would appear just before being placed on the physical medium in Layer 0. The data for Layer 0 starts with the header for Layer 1. The data for Layer 1 starts with the header for Layer 2 and so on. The rules for communication between layers deal mostly with how each layer interprets the data. How closely the layers are tied to each other depends on the cross boundary checking that occurs when the stack is implemented.

To better understand the rules of interaction that exist between layers of a protocol

stack one must consider a hypothetical stack that is built of two layers (Layer A and Layer B). Assuming that Layer A is the lowest layer, the data for Layer A includes the header for Layer B as well as the communication data. How does Layer A know how to parse the data? Does Layer A need to know how to parse Layer B or does it simply pass the information to the upper levels of the stack? Are multiple protocols available for Layer B? If so, how does Layer A know which protocol to handoff to as the data passed back up the stack? To answer some of these questions consider the possible scenarios which determine the interactions between the two layers in the example.

If Layer A is not tied to Layer B there will be no validation or dependence between the two layers. In this case the two layers are said to have “no connection” between each other. The two layers are essentially independent of each other and there is no cross referencing of fields between the two layers. An example of this type of protocol stack would be the standards used to facilitate the propagation of e-mail. The protocol stack used for e-mail consists of two layers, one for the “envelope” that contains the original message and one for the delivery of that envelope. The current implementation is highly flexible, but there is no dependence between the two layers necessary to deliver and read an e-mail message. This loose coupling of the standards used for e-mail has led to the exploitation of these standards known as “Spam” or unsolicited bulk e-mail.

Another possible scenario is if Layer A is loosely tied to Layer B but Layer B is not tied to Layer A. In this case the rules of interaction are only one-way and the two layers would have a “unidirectional” connection. An example of this type of protocol stack is TCP/IP, specifically between Layer 1 (Network) and Layer 2 (Transport). Layer 1 uses a single field to control how the data is interpreted at Layer 2 when it is received. However, there is little validation performed on this interaction until the data is processed by Layer 2 on the receiving end of the communication. This loose coupling of the two layers in the TCP/IP stack provides an avenue of exploitation that is explored later in this paper.

If Layer A is closely tied to Layer B and Layer B is closely tied to Layer A the two layers are said to have a “bidirectional” connection between each other. An example of this type of connection is present in Layer 0 of the TCP/IP standards stack. The media access layer and the physical layers are closely tied to each other. Replacing or changing one of these layers would render the other layer broken or inoperable. For instance, if you switched the physical layer from Ethernet to Token Ring the media access layer for Ethernet might not be able to communicate with the physical layer of Token Ring.

Placing these types of connections on a scale of exploitability where 0 is “not exploitable” and 10 is “exploitable by default” the following results are found. The bidirectional connection could be said to have an exploitable value close to “0” because any change to one

layer of the communication stack could potentially render all communication unusable. The unidirectional connection could be said to have an exploitable value of between “3” and “6” because only a few connection points need to be retained to ensure the successful delivery of the communication. The most exploitable is the “not connected”. The fully disconnected protocol stacks are prone to exploitation because no inherent checking occurs as the data passes between layers. The growth of spam [14] (unsolicited bulk e-mail) is a very good example of how simply, and successfully, a non-connected standards stack can be exploited.

### *Communicating with the Standards Stack*

Once the rules of the protocol stack are determined the final piece of the puzzle is how the operating system’s implementation of the rules allows the program to interact with the protocol stack. An example would be the specific rules for building a communication channel between two entities using TCP/IP for a Linux operating system. This paper will consider the requirements for building communication between two Fedora Core 6 hosts running Linux kernel 2.6.19-1.2895.

To facilitate communications between entities using TCP/IP each program must use sockets for communication [15]. Sockets are a powerful way to configure communication endpoints. The declaration of a new communication endpoint requires three parameters and

returns a descriptor which can be used for communication later [16].

| | |
|---|---|
| *domain* | The communication domain to use for the socket. This paper focuses on the PF_INET (IPv4) domain. This will end up in byte 0 offset from zero of the IP header. |
| *type* | The type of connection to build. SOCK_STREAM is for TCP, SOCK_DGRAM is for UDP and SOCK_RAW is for IP. |
| *protocol* | This is set to 0 (zero) unless the SOCK_RAW type is used. In the case of SOCK_RAW an acceptable protocol number defined in /etc/protocols is accepted. |

Depending on the domain and type of socket created the visibility inside the packet(s) and the rules that are applied to the socket are different. The domain used in this paper is PF_INET which is valid for IPv4. The PF_INET domain of sockets accepts three types of sockets: SOCK_STREAM, SOCK_DGRAM and SOCK_RAW [17]. If a SOCK_STREAM is defined only TCP packets will be intercepted. If a SOCK_DGRAM is defined only UDP packets will be intercepted. For access to all other packets a SOCK_RAW socket must be

defined. If a SOCK_RAW socket is defined the filtering is determined by the protocol. Common options for the protocol field are IPPROTO_IP, IPPROTO_ICMP, IPPROTO_TCP or IPPROTO_UDP. Figure 4 visually depicts where each domain-type combination would attach in the protocol stack.

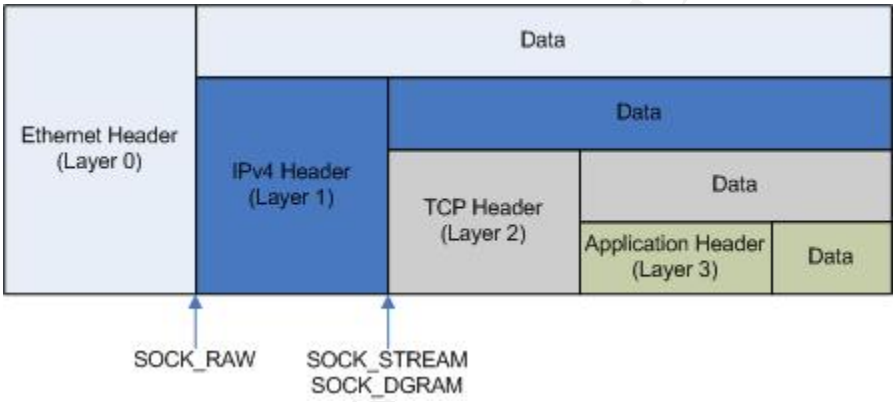


*Figure 4: Socket Domains – TCP/IP Protocol Stack*

In the following example, a raw socket is defined with a filter for only TCP packets.

```
#include <sys/types.h>
#include <sys/socket.h>

int main() {
  int testsocket;
  // open a raw IP socket
  testsocket = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);

// close the TCP socket
  close(testsocket);
}
```

By default PF_INET sockets defined as SOCK_RAW assume that only the upper level

header will be modifiable [18]. To send packets with a modified IP header the socket must be adjusted to allow for passing an IP header as part of the packet definition. Setting the socket option IP_HDRINCL on the defined socket to enable the socket to send the IP header. The following example shows how to setup the socket to send a modified IP header.

```
#include <sys/types.h>
#include <sys/socket.h>

int main() {
  int testsocket;
  int dummy = 1;
  // open a raw IP socket
  testsocket = socket(PF_INET, SOCK_RAW, IPPROTO_TCP);

  // set the socket option to allow the IP header
  setsockopt(server_sockfd, IPPROTO_IP, IP_HDRINCL, &dummy, sizeof(dummy));

// close the TCP socket
  close(testsocket);
}
```

The next step that is sometimes required for socket communication is to bind the socket to a specific address-port combination. For connection-oriented sockets of type SOCK_STREAM the act of binding the socket is required to receive any connections [19]. For connection-less sockets (SOCK_DGRAM) or raw sockets (SOCK_RAW) this restriction does not apply and connections can be received prior to binding the socket. However, the socket will not have authoritative control over that communication until it is bound to a local address [17]. Thus, until the socket is bound any communications destined for that socket will

be handled by protocol stack as a default connection.

The three steps defined above outline the basic communications needed to control the socket and direct the communications with the protocol stack in a controlled manner. Defining the correct domain and type of socket, setting appropriate socket options, and binding the sockets as necessary allows for efficient communication with the communications stack. For sockets, these steps are available regardless of the underlying protocol stack in use or where in the stack the socket is defined. Switching to a protocol stack facilitating IPX/SPX communications is a matter of redefining the type of socket in use.

## 3 TCP/IP Gap Analysis – In-Depth

This paper focuses on the use of the loose coupling in the TCP/IP protocol stack to gain the ability to transparently pass traffic from one host to a second host. This is possible because of the loose coupling between layer 1 (IP) and layer 2 (TCP) in the protocol stack. The flexibility that is provided by the TCP/IP stack as well as the rules for communication with the stack through sockets allows for the possibility of this approach to succeed.

Layer 1 (IP) is tasked with the delivery of the connection from HOSTA to HOSTB using identifiers called IP addresses. Layer 2 (TCP) is tasked with the delivery of the connection to the appropriate application once it arrives at the specified host. Typically both hosts that are a part of the connection connect to the same part of the stack to facilitate the connection and sharing of information. Figure 5 shows the connections of two hosts at the same point in the TCP/IP stack. Both hosts begin sending data just after the TCP header in the scenario depicted in figure 5.

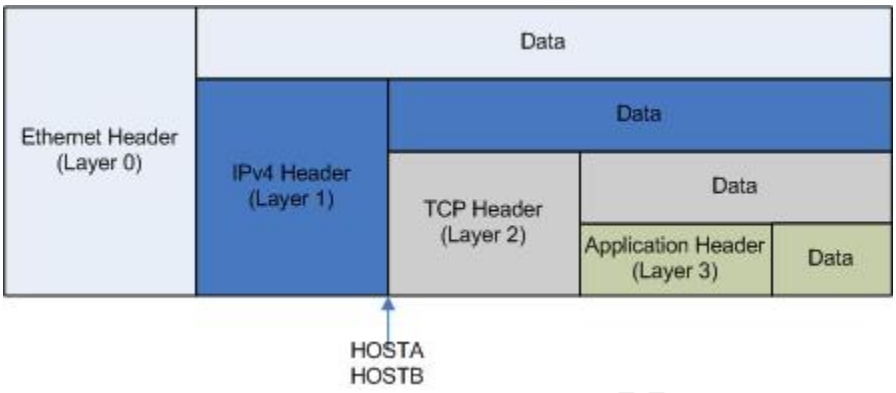


*Figure 5: Normal Host Connections*

However, the connection of both hosts at the same point in the stack is not required. Routers normally only connect up to the boundary between Layer 1 and Layer 0 so they can deliver the packets to the correct destination host. Figure 6 shows the connection difference between the sending host and the delivery router(s) that the packet traverses.

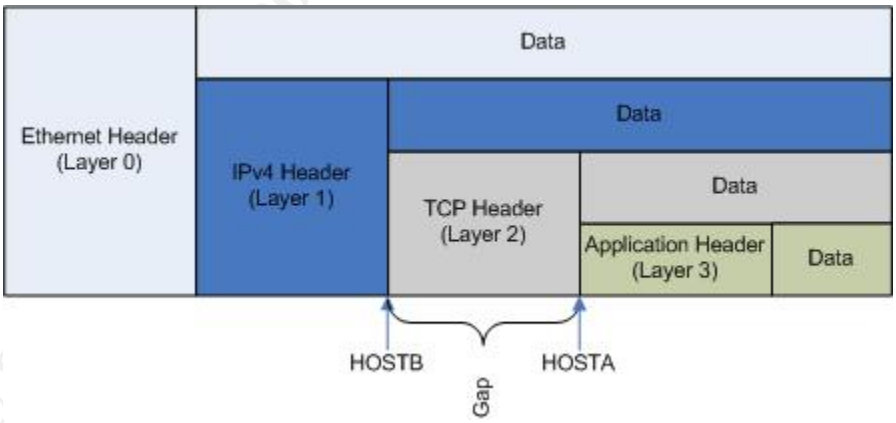


*Figure 6: Subverted Host Connections*

Using this flexibility, the sending and receiving host could attach at different places in

the protocol stack providing a gap between where the expected connection is and where the actual connection occurs. This gap provides the ability for the sending host to pass information to the receiving host and have it successfully pass through the network security layers. This is possible only if the network security countermeasures make an assumption about where the receiving host will connect in the protocol stack. In fact, this assumption is frequently made by security devices and security analysts, myself included (prior to conducting research for this paper of course). The assumption is directed by a single field in the IP header which defines how the upper layer protocol header should be interpreted. This field, byte 9 offset from zero, of the IP header, called the protocol field, is normally used to instruct the receiving host how to interpret the upper layer header [6]. This field effectively provides direction for how to apply a bitmask across a predefined number of bytes before passing the data up the protocol stack. Based on the value of this field the size of the bitmask will vary. This paper takes a close look at this field and demonstrates why the assumption that this field is accurately set in connections is not a safe assumption to make.

### *The Setup*

For this approach to be successful the gap provided by the assumption must be undetectable (ideal) or very hard to discern from normal communications. The first step to hiding this connection is to convince the analyst, or countermeasure, that the receiving host

will connect at a higher point in the protocol stack. The second step is to use enough of the higher protocol header to appear like a normal connection attempt. Finally, an ideal connection will stay within a set of expected characteristics for normal communications. This includes ensuring that the checksums are accurate, ensuring the flags are setup, basically abiding by all the common "red flags" which are used to isolate rogue communications.

For the purposes of this paper the examination of the characteristics of normal communications will start at Layer 1 (IP) and include Layer 2 (TCP). TCP communications will be used because these kinds of connections are commonly allowed through firewalls and other network security layer devices. Both layers will be examined and the normalcy tests that are used to qualify "normal" traffic will be documented.

### *Normal Communications*

First, the normalcy tests for the IP header are documented since this is interpreted first by the receiving host. Normally an IP header will need to have the following fields with common values provided [6]:

| | |
|---|---|
| Version | The version of normal communications should be set to depict IPv4 or IPv6 traffic. Any values outside of these two will be |

| | |
|---|---|
| | unusual and could attract unwanted attention. |
| IP Header Length (IHL) | The IP Header Length is a field which determines how long the IP header is. Unless IP Options are present this is set to five for a twenty byte header length. The presence of any IP options could attract unwanted attention. |
| Total Length | The total length field should be accurate because inaccurate length fields could attract unwanted attention. |
| Flags | The flags should be set to reflect non-fragmented traffic since fragmented traffic is uncommon “in-the-wild” and can attract unwanted attention. |
| Time-To-Live (TTL) | The TTL field should be set to the default for the OS the packet is being sent from. Setting the TTL to uncommon values could be an easy sign of crafting. |
| Protocol | The protocol field should be set to the upper layer protocol in use. |

| | |
|---|---|
| Header Checksum | The checksum should be accurate and functioning or the packet could be dropped before reaching the destination host by an intermediary host. |
| Source Address | The source address should be the address of the sending host and should be a valid IP address for host which is initiating the connection attempt. |
| Destination Address | The destination address should be the address of the receiving host and should be a valid address for the network the packet is traversing. |

Comparing these fields to the full list of fields in the IP header there is very little room for building a gap to disguise communications. In addition, trying to build any gap using this layer would force the communication to a local area network only because the header could be corrupt and not delivered successfully through intermediary routers.

The next layer, TCP is less restrictive in its requirements since there is very little information actually needed to make a successful connection. This is mostly due to the various other protocols available at this layer (UDP, ICMP, etc.) so validations at this layer are

usually held to only a few areas. These checks are defined in the next table [8].

| | |
|---|---|
| Source Port | The source port is needed to provide for identification of the connection attempt. While it is not a common check unusual source ports (low numbers) can attract unwanted attention. |
| Destination Port | The destination port is needed and should adhere to a common communication port. HTTP (80) is effective to use because the normally high volumes of traffic makes the connection harder to distinguish from the noise. Using unusual destination ports could result in a dropped connection and an easily identifiable signature. |
| Flags | The flags need to be preserved and should follow the TCP standard. Unusual combinations of TCP flags are normally associated with crafted packets. |
| Checksum | The checksum should be valid. A bad checksum draws unwanted attention because it is not normal to find a packet with bad checksums unless errors are occurring on the network. |

| | |
|---|---|
| Data on SYN | A SYN packet does not normally contain data. If data is found on a SYN packet the packet would draw unwanted attention. |

The TCP header has far fewer checks that the IP header. In all, only seven out of the twenty bytes commonly found in a TCP packet were needed. That leaves a gap of thirteen bytes for hiding information inside the TCP header. The interesting aspect of using this gap for passing information is that if only SYN packets from the initial communication are used the data would not be preserved once the connection was established.

Another scenario would be to place the data after the TCP packet during a connection attempt. However, this is uncommon in normal connections so data on a SYN attracts attention from analysts. In addition the data is preserved upon connection reconstruction and is easily discernable and is commonly treated as normal data for the connection. Therefore, passing information inside the header is the best chance at staying hidden.

### *The Attempt*

The approach is to use the protocol field in the IP header to provide a gap in the TCP header for hiding a covert channel, or hidden communication channel. This is possible because so many of the security devices explicitly trust this field and rely upon it for further

analysis of the packet. This section attempts to adhere to the normalcy tests defined in the previous section to reduce the chance of discovery.

For this approach, raw sockets on Linux will be utilized to give the control over the packet needed to hide and interpret the data. This section will attempt to embed a two word phrase into a SYN packet. The crafted SYN packet is examined and compared to a real SYN packet side-by-side in the analysis section to determine how successful the covert channel would be. The entire source code used in this example is provided in the appendix for examination.

The first step is to define the specific container that will be used. Looking at the TCP header the container between byte 4 offset from zero and byte 11 offset from zero is utilized in this example. This allows for preservation of the flags and checksum later in the packet. Another two-byte section is available at the end of the TCP header as well (urgent pointer) but this example does not utilize this area for passing information.

The first step is to define the raw socket for the receiving side. The receiving side needs a socket of type SOCK_RAW from the domain PF_INET. The socket does not need to be bound for this approach to work and not binding the socket allows for normal resets to be sent which will further disguise the covert channel. If the socket cannot be constructed then

the program should error out and state why it could not succeed.

```
if ( (server_sockfd = socket(PF_INET, SOCK_RAW, IPPROTO_TCP)) < 0) {
    perror("no good: serverraw");
    return(-1);
  }
```

The sending side of the channel also needs access to a socket of type SOCK_RAW from the domain PF_INET. In addition the sending side needs to be able to send a crafted IP header as part of the SYN packet. Again the program should not attempt to send and error out if either of these two items cannot be completed.

```
  // Open a raw socket to send the packet on
  if ( (server_sockfd = socket(PF_INET, SOCK_RAW, IPPROTO_TCP)) < 0) {
    perror("no good: clientraw");
    return(-1);
  }

  // Set the socket options to allow for raw IP headers
  if ((setsockopt(server_sockfd, IPPROTO_IP, IP_HDRINCL, &dummy, sizeof(dummy))) < 0)
{
    perror("no good: clientraw");
    return(-1);
  }
```

Neither side of this connection attempt needs to bind the socket to an address because the covert channel does not require a complete connection. For this demonstration only a single SYN packet is sent and received. If interaction is required then both sides would need to bind to a socket to be able to interact with the communications correctly.

The sending side needs to define the raw structures to hold the covert channel. This

includes a raw IP header, raw TCP header, and an empty packet to send. Once defined the packet space should be cleared to ensure that only the specified data is present.

```
  // setup raw structures and pointers to the structures
  // first, IP header information
  siph = (struct iphdr *) malloc(sizeof(struct iphdr));
  // second, TCP header information
  stcph = (struct tcphdr *) malloc(sizeof(struct tcphdr));
  // third, packet space... we just need a standard syn packet size
  packet = (char *) malloc((sizeof(struct iphdr) + sizeof(struct tcphdr)) *
sizeof(char));
  // clear the packet space
  memset(packet, '\0', sizeof(packet));
```

Once the raw containers are defined the structures need to be linked on top of the packet area so they can easily be populated with information.

```
  // Link up the pointer structures to their place in the packet
  siph = (struct iphdr *) packet;
  stcph = (struct tcphdr *) (packet + sizeof(struct iphdr));
```

Next a pseudo-header needs to be linked into place so the TCP checksum can be accomplished. This pseudo-header linked in place just before the TCP header to more easily facilitate the calculation of the TCP checksum.

```
  // Now assign up the pseudo header for the tcp checksum
  stoyh = (struct pseuhdr *) (packet + sizeof(struct iphdr) - sizeof(struct
pseuhdr));
```

Then populate the pseudo-header and TCP header with information and calculate the TCP checksum for the crafted packet.

```
  //setup pseudo header
  stoyh->saddr = inet_addr("127.0.0.1");
  stoyh->daddr = inet_addr("127.0.0.1");
  stoyh->protocol = IPPROTO_TCP;
  stoyh->length = htons(sizeof(struct tcphdr));

  //setup tcphdr
  stcph->source = htons(12345);
  stcph->dest = 12345;
  stcph->ack_seq = 0x414C4C20;
  stcph->seq = 0x444F4E45;
  stcph->doff = 5;
  stcph->syn = 0x01;
  stcph->window = htons(2048);
  stcph->check = in_cksum((unsigned short *)stoyh, sizeof(struct pseuhdr) +
sizeof(struct tcphdr));
```

Once the TCP checksum is created the IP header needs to be cleared of the pseudo-header information and repopulated with the information for the crafted IP header.

```
  // clear iphdr of pseudo-header info
  memset(siph, 0x00, sizeof(struct iphdr));
  // setup iphdr
  siph->ihl = 5;
  siph->version = 4;
  siph->tos = 0;
  siph->tot_len = sizeof(struct iphdr) + sizeof(struct tcphdr);
  siph->id = htons(getuid());
  siph->ttl = 255;
  siph->protocol = IPPROTO_TCP;
  siph->saddr = inet_addr("127.0.0.1");
  siph->daddr = inet_addr("127.0.0.1");
  siph->check = in_cksum((unsigned short *)siph, sizeof(struct iphdr));
```

The final step before sending the packet is to define an address structure for the socket in order to send information over it. This address structure is specific to the domain and type of socket created. Once defined the address structure is used to send the packet on

the network. Finally, the socket is closed once the sending is completed.

```
  // setup the socket addressing
  saddr.sin_family = PF_INET;
  saddr.sin_port = IPPROTO_IP;
  saddr.sin_addr.s_addr = inet_addr("127.0.0.1");

  // send the packet
  if ((retval = sendto(server_sockfd, packet, siph->tot_len, 0, (struct sockaddr
*)&saddr, sizeof(saddr))) == -1) {
    log_msg(1, "ERROR: failed to send", "main");
  }
  // close the socket
  close(server_sockfd);
```

This completes the processing for the sending side. The receiving side has a socket created and is ready to wait for the incoming packets. To instruct the socket to wait for an incoming packet the recvfrom() routine is used.

```
server_len = recvfrom(server_sockfd, buffer, 8192, 0, NULL, NULL);
```

Once the packet is received the header structures need to be mapped upon the packet that was received. First the IP header is mapped so it can be parsed.

```
// Map IP Header structure onto receive buffer
iphead = (struct iphdr *)buffer;
```

Second, the IP header is examined to determine where the upper layer protocol header begins and also what the upper layer protocol is. This example is using only TCP packets so these two fields are known and a TCP header can be mapped onto the packet.

```
// Map the TCP Header structure onto the receive buffer
```

```
tcphead = (struct tcphdr *)(buffer+(iphead->ihl*4));
```

When examining these past two steps carefully it becomes evident that the receive buffer is nothing more than a collection of bytes. The act of mapping the structures onto this buffer provides form to the packet allowing it to be analyzed. The normal flow of mapping is the following:

1. Map IP header onto receive buffer
2. Determine end of IP header
3. Determine upper layer protocol (iphdr[9])
4. Map upper layer header onto receive buffer starting at the end of the IP header

However, there is no enforcement that the buffer only contains the upper layer protocol specified by the ninth byte offset from zero of the IP header. A second approach to this scenario would be to use a UDP header instead of the TCP header but leave the IP protocol field designating a TCP header is needed. The normal size of a TCP header (20 bytes) minus the normal size of a UDP header (8 bytes) also reveals that a significant gap can be utilized with this approach. However, this approach will not guarantee an accurate checksum or ensure that the appropriate flags are enforced. These two aspects combined are a

significant red-flag so this approach was abandoned in favor of the approach utilizing the gaps in the TCP header to embed data.

Continuing in the inspection of the received packet, once the TCP header has been mapped on top of the buffer, the hidden content area needs to be mapped onto the buffer area so the information can be extracted.

```
// map the content pointer onto the receive buffer
content = (unsigned char *)(buffer + (iphead->ihl*4) + 4);
```

This defines the receive buffer to begin at the fourth byte offset from zero of the TCP header. This preserves the Source Port and Destination Port fields of the TCP header. The information inside the covert channel can be extracted by referencing the content variable as a character array. In addition the program prints out summary data about the crafted packet including the content that was passed.

```
[INFO]-[main]: ---->
[INFO]-[main]: Caught tcp packet: 40 bytes
[INFO]-[main]: Dumping encoded information.
[INFO]-[print_packet]: IPHDR LEN: 20 bytes
[INFO]-[print_packet]: TCPHDR LEN: 20 bytes
[INFO]-[print_packet]: TCPFLAGS: ----S-
[DEBUG]-[print_packet]: CONTENT: ALL DONE
[INFO]-[main]: ---->
```

From this log trace the crafted packet was 40 bytes in length, 20 bytes for the IP header and 20 bytes for the TCP header. There was no data attached to this packet but the

content “ALL DONE” is extracted from the covert channel.

### *The Analysis*

To analyze the data two SYN packets are compared. The first packet is a normal SYN packet sent to a service to establish a connection. The second packet is the SYN packet crafted in the example above. The two packets are displayed side-by-side in hex for comparison. The packets displayed below contain the Ethernet header so there are an additional 14 bytes of information in this packet. The Ethernet header is highlighted in Blue, the IP headers is highlighted in green and the TCP header is in pink.

| | |
|---|---|
| 0000 00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00<br>0010 00 3c 42 5e 40 00 40 06 fa 5b 7f 00 00 01 7f 00<br>0020 00 01 83 0d 06 26 ad c2 f8 d6 00 00 00 00 a0 02<br>0030 80 18 a1 ea 00 00 02 04 40 0c 04 02 08 0a 03 4f<br>0040 ba 88 00 00 00 00 01 03 03 05 | Normal SYN |
| 0000 00 00 00 00 00 00 00 00 00 00 00 00 08 00 45 00<br>0010 00 28 73 90 00 00 ff 06 4a 3d 7f 00 00 01 7f 00<br>0020 00 01 30 39 39 30 45 4e 4f 44 20 4c 4c 41 50 02<br>0030 08 00 3f 57 00 00 | Crafted SYN |

The normalcy tests from the previous section need to be applied to the crafted SYN packet to determine if this approach is successful in disguising itself as normal traffic. The

following table lists the tests and the status of each one when compared to the crafted SYN packet. First the IP header tests are analyzed.

| | |
|---|---|
| Version | Byte 0 of the IP header is set to “4” so this test passes. |
| IHL | Byte 1 offset from zero of the IP header is set to “5” for a 20 byte IP header so this test passes. |
| Total Length | Bytes 2 and 3 offset from zero is set to 0x0028 or 40 bytes so this test passes. |
| Flags | Byte 6 offset from zero is set to zero so there are no flags set so this test passes. |
| TTL | Byte 8 offset from zero is set to 0xFF (255) which is normal for a Linux host so this test passes. |
| Protocol | Byte 9 offset from zero is set to 0x06 which is TCP and is normal for a SYN packet so this test passes. |
| Header Checksum | The checksum is correct so this test passes. |

| | |
|---|---|
| Source Address | The source address is accurate so this test passes. |
| Destination Address | The destination address is accurate so this test passes. |

The crafted packet passes the IP header normalcy tests but this is not a surprise because this approach does not require any modifications to the IP header. Only the upper layer protocol header is modified. The next step is to compare the TCP header normalcy tests.

| | |
|---|---|
| Source Port | Bytes 0 and 1 offset from zero of the TCP header is set to a valid ephemeral port so this test passes. |
| Destination Port | Bytes 2 and 3 offset from zero of the TCP header is set to an ephemeral port which is unusual and may post problems crossing a firewall. For more success the destination port should be set to a common port such as HTTP (80). |
| Flags | Byte 13 offset from zero of the TCP header contains the TCP flags and only the SYN flag is set so this test passes. |
| Checksum | The checksum is valid for this packet so this test passes. |

| | |
|---|---|
| Data on SYN | No data is present on this SYN packet so this test passes. |

This packet passes all of the defined normalcy tests for the TCP header. Theoretically this approach could be utilized and it would leave no red flags which would distinguish this approach from normal traffic. This is especially true if this approach is combined with a common port like HTTP where the volume of traffic would effectively hide the covert channel.

## 4 Conclusions

In 2002 there were many noted attacks which exploited a similar assumption in perimeter security devices [12]. The description of attacks describes the encapsulation of an IPv6 packet tunneled inside an IPv4 packet. This effectively creates a gap because the countermeasures were developed to assume that an IPv4 header would be present and no signatures were built against an IPv6 header. However, the delivery devices made no such assumption and were able to handle either IPv4 or IPv6 packets meaning that, because the signatures did not match, the gap introduced by the IPv6 header enabled virtually any attack to pass undetected. The idea is to evade detection by developing a gap between where the initiating program begins interpretation and where the countermeasures or intermediary hosts begin interpretation.

The assumption discussed in this paper is that the information contained in the protocol field of the IP header is accurate and can be trusted. However, as this paper shows, this assumption is not always an accurate one. Due to the lack of a bidirectional connection between Layer 1 (IP) and Layer 2 (TCP) of the protocol stack the information cannot be trusted. This is not to say that the presence of a bidirectional connection is guaranteed to result in a trustworthy connection but the additional checks in a bidirectional connection make the information more trustworthy.

The approach discussed in this paper relies on the preservation of the upper layer protocol header. This reliance on the preservation of information for transmission provides characteristics which make this approach difficult to implement. First, any perimeter security countermeasure which rebuilds the connection at or above the upper layer protocol header will erase this covert channel in both directions. Second, due to the nature of the TCP protocol once a connection is established the portions of the header being used for the channel would need to be preserved to match the normalcy tests of a complete TCP connection. This approach would still work and could be used to embed a chat program inside a HTTP communication such that browsing various web pages would pass information on each SYN packet only.

The gaps introduced in loosely connected protocols are incredibly powerful tools for

evading detection and will always be available unless future protocol stacks are built with bidirectional connections in mind. Until the stronger connections are available and multiple checks present in the protocol stacks it is recommended that a form of application-level countermeasure, such as a proxy-level firewall, be utilized as part of the perimeter security. The use of a proxy-level countermeasure will enforce the point of interpretation for packets entering and departing the perimeter. This would be the only effective way of preventing this approach from being successful.

## 5 References

[1] Stevens, W. Richard (2000). TCP/IP Illustrated Volume 1: The Protocols. New Jersey, Addison Wesley

[2] Picture of OSI Model. Retrieved January 10, 2007 from http://upload.wikimedia.org/wikipedia/en/thumb/f/ff/Osi_model_trad.jpg/450px-Osi_model_trad.jpg

[3] Picture of OSI Model compared to TCP/IP Layers. Retrieved January 10, 2007 from http://whatis.techtarget.com/digitalguide/images/Misc/fsimage2a.jpg

[4] IEEE 802.3. Wikipedia, version updated on February 3rd, 2007. Retrieved February 10th, 2007 from http://en.wikipedia.org/wiki/IEEE_802.3

[5] Hornig, Charles (April 1984). RFC 894 (RFC894). Internet RFC/STD/FYI/BCP Archives. Retrieved February 10th, 2007 from http://www.faqs.org/rfcs/rfc894.html

[6] Information Sciences Institute (1981). Internet Protocol Darpa Internet Program Protocol Specification. Internet RFC/STD/FYI/BCP Archives. Retrieved February 10th, 2007 from http://www.faqs.org/rfcs/rfc791.html

[7] Postel, J. (August 1980). RFC 768 (RFC768). Internet RFC/STD/FYI/BCP Archives. Retrieved February 10th, 2007 from http://www.faqs.org/rfcs/rfc768.html

[8] Information Sciences Institute (1981). Transmission Control Protocol Darpa Internet

Program Protocol Specification. Internet RFC/STD/FYI/BCP Archives. Retrieved February 10th, 2007 from http://www.faqs.org/rfcs/rfc793.html

[9] Network Working Group (1999). Hypertext Transfer Protocol – HTTP/1.1. Internet RFC/STD/FYI/BCP Archives. Retrieved February 10th, 2007 from http://www.faqs.org/rfcs/rfc2616.html

[10] Postel, Jonathan B (1982). Simple Mail Transfer Protocol. Internet RFC/STD/FYI/BCP Archives. Retrieved February 10th, 2007 from http://www.faqs.org/rfcs/rfc821.html

[11] Crocker, David H. (1982). Standard for the Format of ARPA Internet Text Messages. Internet RFC/STD/FYI/BCP Archives. Retrieved February 10th, 2007 from http://www.faqs.org/rfcs/rfc822.html

[12] Spitzer, Lance (Dec. 17, 2002). IPv6. Honeypots mailing list @ Honeypots.org. Retrieved February 10th, 2007 from http://seclists.org/honeypots/2002/q4/0105.html

[13] Standards Stack, Wikipedia, version updated on February 6th, 2007. Retrieved February 10th, 2007 from http://en.wikipedia.org/wiki/Protocol_stack

[14] Robinson, Donald (Jan. 12, 2007). Spam volume to double soon. IT Week. Retrieved February 10th, 2007 from http://www.itweek.co.uk/itweek/news/2172344/spam-volume-double-soon

[15] Berkley sockets, Wikipedia, version updated on February 9th, 2007. Retrieved Febryary 10th, 2007 from http://en.wikipedia.org/wiki/Berkeley_sockets

[16] Socket(2) manpage, die.net. Retrieved on February 10th, 2007 from http://www.die.net/doc/linux/man/man2/socket.2.html

[17] ip(7) manpage, die.net. Retrieved on February 10th, 2007 from http://www.die.net/doc/linux/man/man7/ip.7.html

[18] raw(7) manpage, die.net. Retrieved on February 10th, 2007 from http://www.die.net/doc/linux/man/man7/raw.7.html

[19] bind(2) manpage, die.net. Retrieved on February 10th, 2007 from http://www.die.net/doc/linux/man/man2/bind.2.html

[20] 10om – member of the excluded team, www.milw0rm.com. Retrieved on February 10th, 2007 from http://www.milw0rm.com/papers/54

# 6 Appendix A: Source Code – clientraw.c

```
/*
 * Copyright (c) 2007
 *     Rodney Caudle.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *     This product includes software developed by Rodney Caudle.
 * 4. Neither the name of the developers may be used to endorse or promote
 *    products derived from this software without specific prior written
 *    permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *     @(#)clientraw.c     0.1 (Rodney Caudle) 02/06/2007
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <arpa/inet.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <netpacket/packet.h>
#include <net/ethernet.h> /* the L2 protocols */
#include <string.h>
#include <malloc.h>

/*
 * This pseudo-header is used for calculating the tcp checksum
 * of the raw packet.
 */
struct pseuhdr {
  unsigned long saddr;
  unsigned long daddr;
  char useless;
  unsigned char protocol;
  unsigned short length;
};

/*
 * This checksum function is used to calculate both the ip checksum and
 */
unsigned short in_cksum(unsigned short *ptr, int nbytes) {
  register long    sum;
  u_short oddbyte;
  register short answer;
  sum = 0;
  while(nbytes > 1) {
    sum += *ptr++;
    nbytes -= 2;
  }
  if(nbytes == 1) {
    oddbyte = 0;
    *((u_char *) &oddbyte) = *(u_char *)ptr;
    sum += oddbyte;
  }
  sum  = (sum >> 16) + (sum & 0xffff);
  sum += (sum >> 16);
  answer = ~sum;
  return(answer);
}
```

```
void log_msg(int level, char *msg, char *loc) {
  switch(level) {
    case 0:
      printf("[DEBUG]-[%s]: %s\n", loc, msg);
      break;
    default:
      printf("[INFO]-[%s]: %s\n", loc, msg);
  }
  return;
}

int main() {
  int server_sockfd;
  char *packet;           // sending packet
  unsigned char *tcpflags, *content;
  struct iphdr *siph;   // sending ip header
  struct tcphdr *stcph;   // sending tcp header
  struct pseuhdr *stoyh; // sending pseudo header for checksums
  struct sockaddr_in saddr; // sending address for socket
  int retval, dummy = 1;
  char tmp[512];
  char blank[2] = " ";

  // Open a raw socket to send the packet on
  if ( (server_sockfd = socket(PF_INET, SOCK_RAW, IPPROTO_TCP)) < 0) {
    perror("no good: clientraw");
    return(-1);
  }

  // Set the socket options to allow for raw IP headers
  if ((setsockopt(server_sockfd, IPPROTO_IP, IP_HDRINCL, &dummy, sizeof(dummy))) < 0)
{
    perror("no good: clientraw");
    return(-1);
  }

  log_msg(1, "---->", "main");
  log_msg(1, "Sending a packet with the string \"ALL DONE\".", "main");

  // setup raw structures and pointers to the structures
  // first, IP header information
  siph = (struct iphdr *) malloc(sizeof(struct iphdr));
  // second, TCP header information
```

```
  stcph = (struct tcphdr *) malloc(sizeof(struct tcphdr));
  // third, packet space... we just need a standard syn packet size
  packet = (char *) malloc((sizeof(struct iphdr) + sizeof(struct tcphdr)) *
sizeof(char));
  // clear the packet space
  memset(packet, '\0', sizeof(packet));

  // Link up the pointer structures to their place in the packet
  siph = (struct iphdr *) packet;
  stcph = (struct tcphdr *) (packet + sizeof(struct iphdr));

  // Now assign up the pseudo header for the tcp checksum
  stoyh = (struct pseuhdr *) (packet + sizeof(struct iphdr) - sizeof(struct
pseuhdr));

  //setup pseudo header
  stoyh->saddr = inet_addr("127.0.0.1");
  stoyh->daddr = inet_addr("127.0.0.1");
  stoyh->protocol = IPPROTO_TCP;
  stoyh->length = htons(sizeof(struct tcphdr));

  //setup tcphdr
  stcph->source = htons(12345);
  stcph->dest = 12345;
  stcph->ack_seq = 0x414C4C20;
  stcph->seq = 0x444F4E45;
  stcph->doff = 5;
  stcph->syn = 0x01;
  stcph->window = htons(2048);
  stcph->check = in_cksum((unsigned short *)stoyh, sizeof(struct pseuhdr) +
sizeof(struct tcphdr));

  memset(siph, 0x00, sizeof(struct iphdr));
  // setup iphdr
  siph->ihl = 5;
  siph->version = 4;
  siph->tos = 0;
  siph->tot_len = sizeof(struct iphdr) + sizeof(struct tcphdr);
  siph->id = htons(getuid());
  siph->ttl = 255;
  siph->protocol = IPPROTO_TCP;
  siph->saddr = inet_addr("127.0.0.1");
  siph->daddr = inet_addr("127.0.0.1");
```

```
  siph->check = in_cksum((unsigned short *)siph, sizeof(struct iphdr));

  // setup the socket addressing
  saddr.sin_family = PF_INET;
  saddr.sin_port = IPPROTO_IP;
  saddr.sin_addr.s_addr = inet_addr("127.0.0.1");

  // send the packet
  if ((retval = sendto(server_sockfd, packet, siph->tot_len, 0, (struct sockaddr
*)&saddr, sizeof(saddr))) == -1) {
    log_msg(1, "ERROR: failed to send", "main");
  }
  close(server_sockfd);
}
```

# 7 Appendix B: Source Code – serverraw.c

```
/*
 * Copyright (c) 2007
 *     Rodney Caudle.  All rights reserved.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 *    notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 *    notice, this list of conditions and the following disclaimer in the
 *    documentation and/or other materials provided with the distribution.
 * 3. All advertising materials mentioning features or use of this software
 *    must display the following acknowledgement:
 *     This product includes software developed by Rodney Caudle.
 * 4. Neither the name of the developers may be used to endorse or promote
 *    products derived from this software without specific prior written
 *    permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED.  IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
 * FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
 * DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
 * OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
 * HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
 * LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
 * OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
 * SUCH DAMAGE.
 *
 *     @(#)serverraw.c     0.1 (Rodney Caudle) 02/06/2007
 */
#include <sys/types.h>
#include <sys/socket.h>
#include <stdio.h>
#include <unistd.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
#include <arpa/inet.h>
#include <netinet/ip.h>
#include <netinet/tcp.h>
#include <netpacket/packet.h>
#include <net/ethernet.h> /* the L2 protocols */
#include <string.h>
#include <malloc.h>

/*
 * This logging routine is used to log simple messages to
 * STDOUT.
 */
void log_msg(int level, char *msg, char *loc) {
  switch(level) {
    case 0:
      printf("[DEBUG]-[%s]: %s\n", loc, msg);
      break;
    default:
      printf("[INFO]-[%s]: %s\n", loc, msg);
  }
  return;
}

/*
 * print_packet is the main printing routine which will
 * print out the information contained in the incoming
 * packet.
 */
void print_packet(struct iphdr *iph, struct tcphdr *tcph, unsigned char *content, int
server_length) {
  char tmp[512];
  char blank[2] = " ";
  int iplen = iph->ihl * 4;
  int tcplen = tcph->doff * 4;
  int content_length;
  int i;

  strcpy(tmp,blank);
  snprintf(tmp, 512, "IPHDR LEN: %d bytes", iplen);
  log_msg(1, tmp, "print_packet");
  // tcp header length is stored in the lower nibble of byte 12 offset from zero of
the tcp header
  // Need to mask the value and divide it by 4
```

```
  strcpy(tmp,blank);
  snprintf(tmp, 512, "TCPHDR LEN: %d bytes", tcplen);
  log_msg(1, tmp, "print_packet");

  // check for a SYN packet
  strcpy(tmp,blank);
  snprintf(tmp, 512,"TCPFLAGS:       ");
  if ( ((tcph->urg) > 0) ) { tmp[10] = 'U'; } else { tmp[10] = '-'; }
  if ( ((tcph->ack) > 0) ) { tmp[11] = 'A'; } else { tmp[11] = '-'; }
  if ( ((tcph->psh) > 0) ) { tmp[12] = 'P'; } else { tmp[12] = '-'; }
  if ( ((tcph->rst) > 0) ) { tmp[13] = 'R'; } else { tmp[13] = '-'; }
  if ( ((tcph->syn) > 0) ) { tmp[14] = 'S'; } else { tmp[14] = '-'; }
  if ( ((tcph->fin) > 0) ) { tmp[15] = 'F'; } else { tmp[15] = '-'; }
  log_msg(1, tmp, "print_packet");

  // Now that we've dumped the important information from the packet,
  // dump the encoded message.
  // set it to just past the DestPort field of the "tcp" header
  //    ((iph->ihl*4)+16)
  snprintf(tmp, 512, "CONTENT: %c%c%c%c%c%c%c%c%c",
             content[7], content[6],
             content[5], content[4],
             content[3], content[2],
             content[1], content[0]);
  log_msg(0, tmp, "print_packet");

  return;
}

int main() {
  int server_sockfd;
  char buffer[8192];
  unsigned char *tcpflags;
  int server_len;
  struct iphdr *iphead;
  struct tcphdr *tcphead;
  char tmp[512];
  char blank[2] = " ";
  unsigned char *con;

  if ( (server_sockfd = socket(PF_INET, SOCK_RAW, IPPROTO_TCP)) < 0) {
    perror("no good: serverraw");
    return(-1);
```

```
  }

  while (1) {
    log_msg(1, "---->", "main");
    server_len = recvfrom(server_sockfd, buffer, 8192, 0, NULL, NULL);

    // setup the IP header, byte 14 offset from 0 of the packet
    // with SOCK_DGRAM instead of SOCK_RAW there's no Ethernet Header to worry about
so add 14
    iphead = (struct iphdr *)buffer;

    // determine if it is a TCP packet
    if (iphead->protocol == IPPROTO_TCP) {
      snprintf(tmp, 512, "Caught tcp packet: %d bytes", server_len);
      log_msg(1, tmp, "main");

      // setup the TCPHeader, byte (14 + (iphead[0]&0x0F) *4)
      // with SOCK_DGRAM instead of SOCK_RAW there's no Ethernet Header to worry
about so add 14
      tcphead = (struct tcphdr *)(buffer+(iphead->ihl*4));

      // respond to a syn packet; Syn only
      if ( (!tcphead->ack && tcphead->syn) ) {
        // Open some debug
        log_msg(1, "Dumping encoded information.", "main");
        con = (unsigned char *)(buffer + (iphead->ihl*4) + 4);
        print_packet(iphead, tcphead, con, server_len);
      }
    } else {
      strcpy(tmp,blank);
      snprintf(tmp, 512, "Skipped packet: %d bytes", server_len);
      log_msg(0, tmp, "main");
    }
  }
  close(server_sockfd);
}
```

Last Updated: April 23rd, 2018

# Upcoming Training


| | | | |
|---|---|---|---|
| SANS vLive - SEC503: Intrusion Detection In-Depth | SEC503 - 201805, | May 02, 2018 - Jun 14, 2018 | vLive |
| Community SANS Virginia Beach SEC503 | Virginia Beach, VA | May 07, 2018 - May 12, 2018 | Community SANS |
| SANS Security West 2018 | San Diego, CA | May 11, 2018 - May 18, 2018 | Live Event |
| Mentor Session - SEC503 | Houston, TX | Jun 18, 2018 - Jul 18, 2018 | Mentor |
| SANS Oslo June 2018 | Oslo, Norway | Jun 18, 2018 - Jun 23, 2018 | Live Event |
| Minneapolis 2018 - SEC503: Intrusion Detection In-Depth | Minneapolis, MN | Jun 25, 2018 - Jun 30, 2018 | vLive |
| SANS Minneapolis 2018 | Minneapolis, MN | Jun 25, 2018 - Jun 30, 2018 | Live Event |
| SANS London July 2018 | London, United Kingdom | Jul 02, 2018 - Jul 07, 2018 | Live Event |
| SANSFIRE 2018 | Washington, DC | Jul 14, 2018 - Jul 21, 2018 | Live Event |
| Security Operations Summit & Training 2018 | New Orleans, LA | Jul 30, 2018 - Aug 06, 2018 | Live Event |
| SANS San Antonio 2018 | San Antonio, TX | Aug 06, 2018 - Aug 11, 2018 | Live Event |
| San Antonio 2018 - SEC503: Intrusion Detection In-Depth | San Antonio, TX | Aug 06, 2018 - Aug 11, 2018 | vLive |
| Community SANS Columbia SEC503 | Columbia, MD | Aug 13, 2018 - Aug 18, 2018 | Community SANS |
| SANS Virginia Beach 2018 | Virginia Beach, VA | Aug 20, 2018 - Aug 31, 2018 | Live Event |
| SANS Amsterdam September 2018 | Amsterdam, Netherlands | Sep 03, 2018 - Sep 08, 2018 | Live Event |
| SANS Tokyo Autumn 2018 | Tokyo, Japan | Sep 03, 2018 - Sep 15, 2018 | Live Event |
| SANS London September 2018 | London, United Kingdom | Sep 17, 2018 - Sep 22, 2018 | Live Event |
| SANS Network Security 2018 | Las Vegas, NV | Sep 23, 2018 - Sep 30, 2018 | Live Event |
| SANS Brussels October 2018 | Brussels, Belgium | Oct 08, 2018 - Oct 13, 2018 | Live Event |
| SANS Northern VA Fall- Tysons 2018 | Tysons, VA | Oct 13, 2018 - Oct 20, 2018 | Live Event |
| SANS Denver 2018 | Denver, CO | Oct 15, 2018 - Oct 20, 2018 | Live Event |
| SANS October Singapore 2018 | Singapore, Singapore | Oct 15, 2018 - Oct 28, 2018 | Live Event |
| Mentor Session - SEC503 | Ballston, VA | Nov 01, 2018 - Dec 06, 2018 | Mentor |
| SANS Dallas Fall 2018 | Dallas, TX | Nov 05, 2018 - Nov 10, 2018 | Live Event |
| SANS San Diego Fall 2018 | San Diego, CA | Nov 12, 2018 - Nov 17, 2018 | Live Event |
| SANS Stockholm 2018 | Stockholm, Sweden | Nov 26, 2018 - Dec 01, 2018 | Live Event |
| SANS OnDemand | Online | Anytime | Self Paced |
| SANS SelfStudy | Books & MP3s Only | Anytime | Self Paced |