# Introduction

IPTables has always been a relatively flexible and modular firewall; if it can't currently test for a particular packet characteristic, you have the option of writing a test or modifying an existing test. The catch − as with so much of open source − is that most of us aren't programmers.

It turns out we do have an option that doesn't require programming. Don Cohen was kind enough to write an IPTables module that pulls any bytes you'd like out of the packet, does some manipulation, and sees if the result is in a particular range. For example, I can grab the Fragmentation information out of the IP header, throw away everything except the More Fragments flag, and see if that flag is set.

Without writing any C code at all. :−)

What I'll do is introduce the core concepts here, and put in hopefully enough annotated examples that you'll be able to write your own tests.

I won't be focusing on *what* these fields are, or why you'd want to test them; there are lots of (warning − shameless plug for my employer ahead!) resources for doing that. If you simply need a quick reference for the packet headers, see tcpip.pdf.

All byte positions in this article start counting at 0 as the first byte of the header. For example, in the IP header, byte "0" holds the 4 bit "Version" and 4 bit "IP Header Length", byte "1" holds the "TOS" field, etc.

## Check the value of a 2 byte field

In it's simplest form, u32 grabs a block of 4 bytes starting at Start, applies a mask of Mask to it, and compares the result to Range. Here's the syntax we'll use for our first examples:

```
iptables −m u32 −−u32 "Start
```

We'll generally pick a "Start" value that's 3 less than the last byte in which you're interested. So, if you want bytes 4 and 5 of the IP header (the IP ID field), Start needs to be 5−3 = 2. Mask strips out all the stuff you don't want; it's a bitmask that can be as large as 0xFFFFFFFF. To get to our target of bytes 4 or 5, we have to discard bytes 2 and 3. Here's the mask we'll use: 0x0000FFFF . We'll actually use the shorter, and equivalent, 0xFFFF instead.

So, to test for **IPID**'s from 2 to 256, the iptables command line is:

```
iptables −m u32 −−u32 "20"
```

To read this off from left to right: "Load the u32 module, and perform the following u32 tests on this packet; grab the 4 bytes starting with byte 2 (bytes 2 and 3 are the Total Length field, and bytes 4 and 5 are the IPID), apply a mask of 0x0000FFFF (which sets the first two bytes to all zeroes, leaving the last two bytes untouched), and see if that value − the IPID − falls between 2 and 256 inclusive; if so, return true, otherwise false."

There is no standalone IPID check in IPTables, but this is the equivalent of the "ip[2:2] >= 2 and ip[2:2] <= 256" tcpdump/bpf filter.

I leave off actions in these examples, but you can add things like:

```
−j LOG −−log−prefix "ID−in−2−256 "
−j DROP
```

or any other action. You can also add other tests, as we'll do in a minute.

Don offers this test to see if the **total packet length** is greater than or equal to 256. The total length field is bytes 2 and 3 of the IP header, so our starting position is 3−3 = 0. Since we're pulling out two bytes again, the mask will be 0xFFFF here as well. The final test is:

```
iptables −m u32 −−u32 "0FFF"
```

This is the same as:

```
iptables −m length −−length 256:65535
or the bpf filter
"len >= 256"
```

## Check the value of a 1 byte field

Much the same, except we'll use a mask of 0x000000FF (or it's shorter equivalent 0xFF) to pull out a single byte from the 4 bytes u32 initially hands us. Let's say I want to test the **TTL** field for TTL's below 3 to find people tracerouting to us. Yes, there's a ttl module, but let's see how this would be done in u32.

I want to end up with byte 8 of the IP header, so my starting position is 8−3 = 5. Here's the test:

```
iptables −m u32 −−u32 "5
```

Which is equivalent to:

```
iptables −m ttl −−ttl−lt 4
or the bpf filter
"ip[8] <= 3"
```

## Looking at 4 bytes at once

To check a complete destination IP address, we'll inspect bytes 16−19. Because we want all 4 bytes, we don't need a mask at all. Let's see if the destination address is 224.0.0.1:

```
iptables −m u32 −−u32 "16=0xE0000001"
```

This is equivalent to:

```
iptables −d 224.0.0.1/32
```

If we only want to look at the first three bytes (to check if a source address is part of a given class C network), we'll need to use a mask again. The mask we'll use is 0xFFFFFF00 , which throws away the last octet. Let's check if the source address (from bytes 12−15, although we'll ignore byte 15 with the mask) is in the class C network 192.168.15.0 (0xC0A80F00):

```
iptables −m u32 −−u32 "1280F00"
```

Which is the same as:

```
iptables −s 192.168.15.0/24
```

## Inspecting early bytes in the header

Obviously, if I want to look at the **TOS** field (byte 1 of the IP header), I can't start at byte 1−3 = −2. What we'll do instead is start at byte 0, pull out the byte we want, and then move it down to the last position for easy testing. This isn't the only way we could do this, but it helps demonstrate a technique we'll need in a minute.

To pull out the TOS field, I first ask u32 to give me bytes 0−3 by using an offset of 0. Now, I pull out byte 1 (the second byte in that block) with a mask of 0x00FF0000 . I need to shift the TOS value down to the far right position for easy comparison. To do this, I use a technique called, unsuprisingly, "right shift". The symbol for right shift is ">>"; this is followed by the number of bits right to move the data. If you're unfamiliar with right shift, take a look at this tutorial from Harper College.

I want to move TOS two bytes − or 16 bits − to the right. This is done with ">>16". Now that we have TOS in the correct position, we compare it to 0x08 (Maximize Throughput):

```
iptables −m u32 −−u32 "0x08"
```

which is the equivalent of:

```
iptables −m ttl −−tos 8
```

## Inspecting individual bits

I'd like to look at the "**More Fragments**" flag − a flag which has no existing test in iptables (−f matches 2nd and further fragments, I want to match all fragments except the last). Byte 6 contains this, so I'll start with offset 3 and throw away bytes 3−5. Normally this would use a mask of 0x000000FF, but I also want to discard the other bits in that last byte. The only bit I want to keep is the third from the top (0010 0000), so the mask I'll use is 0x00000020 . Now I have two choices; move that bit down to the lowest position and compare, or leave it in its current position and compare.

To move it down, we'll right shift 5 bits. The final test is:

```
iptables −m u32 −−u32 "3
```

If I take the other approach of leaving the bit where it is, I need to be careful about the compare value on the right. If that bit is turned on, the compare value needs to be 0x20 as well.

```
iptables −m u32 −−u32 "3
```

Both approaches return true if the More Fragments flag is turned on.

## Combining tests

If you want to inspect more than one aspect of a packet, use:

&

between each test.

## Moving on to the TCP header

This is a little tricky. Let's say I'd like to look at bytes 4−7 of the TCP header (the **TCP sequence number**). Let's take the simple approach first, and then look at some ways to improve this.

For our first version, let's assume that the IP header is 20 bytes long − usually a good guess. Our starting point is byte 4 of the tcp header that immediately follows the IP header. Our simplistic test for whether the sequence number is 41 (hex 29) might look like this:

```
iptables −m u32 −−u32 "24=0x29"
```

For packets where the IP header length is 20, this will actually work, but there are a few problems. Let's fix them one by one.

First, we never check to see if the packet is even a TCP packet. This is stored in byte 9 of the IP header, so we'll pull 4 bytes starting at byte 6, drop 6−8, and check to see if it's 6. The new rule that first checks if this is a TCP packet at all and also checks that the Sequence Number is 41 is:

```
iptables −m u32 −−u32 "6&24=0x29"
```

The second problem we've momentarily ignored is the IP header length. True, it usually is 20 bytes long, but it *can* be longer, if IP options are used.

Here are the steps. We pull the IP header length (a nibble that shows how many 4 bytes words there are in the header, usually 5) out of the IP header. We multiply it by 4 to get the number of bytes in the IP header. We use this number to say how many bytes to jump to get to the beginning of the TCP header, and jump 4 more bytes to get to the Sequence number.

To get the header length, we need the first byte: `"0>>24"`, but we need to only grab the lower nibble and we need to multiply that number by 4 to get the actual number of bytes in the header. To do the multiply, we'll right shift 22 instead of 24. With this shift, we'll need to use a mask of 0x3C instead of the 0x0F we would have used. The expression so far is: `"0>>22 On an IP header with no options, that expression returns 20; just what we'd expect. Now we need to tell u32 to` *`use`* `that number and make a jump that many bytes into the packet, a step performed by the "@" operator.`

```
iptables −m u32 −−u32 "6&0>>22
```

The "@" grabs the number we created on its left (20, normally) and jumps that many bytes forward (we can even do this more than once − see the TCP payload section below). The 4 to its right tells u32 to grab bytes 4−7, but u32 knows to pull them relative to the 20 bytes it skipped over. This gives us the Sequence Number, even if the IP header grows because of options. *phew*!

The last quirk to handle is fragments. When we were only working with the IP header, this wasn't an issue; IP is designed in such a way that the IP header itself can never be fragmented. The TCP header and application payload technically might be, and if we're handed the second or further fragment, we might be looking not at the Sequence Number in bytes 4−7, but perhaps some other part of the TCP header, or more likely, some application layer data.

What we'll do is check that this is the first fragment (or an unfragmented packet, the test won't care), so that we're sure we're looking at tcp header info. To do this, we test the fragment offset in most (we discard the top three flag bits) of bytes 6 and 7 of the IP header to make sure the offset is 0. The test is: `"4`

The final expression (check for TCP, check for unfragmented packet or first fragment, and jump over the IP header, checking that bytes 4−7 of the TCP header are equal to 41) is:

```
iptables −m u32 −−u32 "6&4&0>>22
```

If the packet is, in fact, fragmented, we have one more consideration; the fragment might be so small that the field we're testing might have been put in a future fragment! In this one case, it's not an issue because every IP link should handle packets of at least 68 bytes; even if the IP header was at its maximum of 60 bytes, the first 8 bytes of the TCP header should be included in that first fragment.

When we start testing for things further in to the packet, we'll have to depend on u32's ability to simply return false if we ever try to ask for a value that falls outside of the packet being inspected.

## Checking for values in the ICMP header

Let's look for ICMP Host Unreachables (ICMP, type 3, code 1). Just as in the above example, we need to check for the Protocol field (Protocol 1 = ICMP this time) and that we're looking at a complete packet or at least the first fragment: `"6&4`

To check for the ICMP Type and Code, we skip over the IP header again (`"0>>22). To grab the first two bytes, we'll start at offset 0 and just right shift 16 bits. The final test is:`

```
iptables −m u32 −−u32 "6&4&0>>221"
```

## Checking for values in the UDP payload

Lets try going all the way into the packet payload now, and match packets that are UDP DNS queries. Here we're not only going to check for destination port 53, but we're also going to test the top bit of byte 2 of the payload; if set, this is a DNS query.

We start by checking that this is a UDP packet: `"6 We add the now familiar check for first fragment: "4.`

To test the destination port, we grab bytes 2 and 3 from the udp header (after jumping over the IP header as in the previous examples): `"0>>22"`.

If the packet has passed all of the above, we go back to check the payload (remember we have to jump over the variable−length IP and 8 byte UDP headers `"0>>22..."`) to make sure this is a DNS *query* rather than a response. To grab the high bit from byte 2, I'll use offset 8 to grab the first 4 payload bytes, right shift 15 bits to deposit the Query bit in the lowest position, and throw away all the rest of the bits with a mask of 0x01: `"0>>221"`

The final test is:

```
iptables −m u32 −−u32 "6&4&0>>22 &0>>221"
```

Ugh. I've seen stellar noise that had less entropy :−) Note that we're doing the whole thing with u32 checks; we could pull out the "udp", "first/no fragment" and "port 53" checks into other modules, and end up with this slightly more readable version:

```
iptables −p udp −−dport 53 \! −f −m u32 −−u32 "0>>221"
```

## Checking for values in the TCP payload

As with the udp payload example above, this approach is only useful if we're absolutely certain of where our data of interest can be found. In this example, I want to find ssh sessions, even if they're on a port _other_ than 22. But wait − ssh connections are encrypted! Sounds hopeless? Actually, it isn't.

The first thing to happen in an ssh connection is that the server sends a protocol string back to the client. The protocol looks like: `SSH−protoversion−softwareversion comments`

The protoversion is generally 1.99, 2.0, or 1.5 . The entire line, including a possible trailing carriage return and linefeed, needs to be smaller than 255 bytes. This line _may_ have another line before it, but this would break compatibility with ssh 1.0 clients, so I'm guessing this is uncommon. The above information comes from draft−ietf−secsh−transport−17.txt , if you want to see more details.

The interesting piece, as far as we're concerned, is that "SSH−" string. It lands in the first 4 bytes of the connection. Bingo! We can use the u32 module to look at only those bytes in the early part of a connection; this is a much less intensive check than checking with the string module, and makes it feasible to look for ssh connections on any port without undue load on the firewall.

Lets get the easy stuff out of the way. I want to check tcp packets, unfragmented ones, I'm only interested in packets in the first 255 bytes of a connection, and established ones, obviously, and the entire packet length needs to be between 45 and 375 bytes (think about the minimums and maximums for IP and tcp headers and ssh protocol string):

```
iptables −p tcp \! −f −m connbytes −−connbytes 0:255 −m state −−state
ESTABLISHED −m length −−length 46:375
```

With these restrictions, we should be inspecting very few packets; this allows us to look at the first few packets of every connection for legitimate or rogue ssh connections.

Now u32 does its magic. We'll build it up from scratch. Because we've already checked for protocol TCP, we'll skip 6 We skip over the IP header with `0>>22> 0>>22`

We now need to skip over the tcp header; we pull the TCP header length from the first half of byte 12. Although we'd need to right shift it 28 bits to get it into the last 4 bits of our 32 bit buffer, we have to multiply this by 4, too, to get the number of 4−byte words this number expresses; this is why we right shift by 26 and mask again with 0x3C:

```
0>>2212>>26
```

This slides us into the tcp payload portion of the packet. As the bytes we want to test should generally be in the first 4 bytes of the packet, we just grab 4 bytes starting at 0 and compare them to 0x5353482D (the hex equivalent of "SSH−"):

```
0>>2212>>260=0x5353482D
```

Here's the entire test, line wrapped to save screen space:

```
iptables −p tcp \! −f −m connbytes −−connbytes 0:255 −m state −−state
ESTABLISHED −m length −−length 46:375 −m u32 −−u32
"0>>2212>>260=0x5353482D"
```

Once I've done this, I can even check for specific SSH protocol versions with `−m string −−string "SSH−1.99"` or even as specific as `−m string −−string "SSH−1.99−OpenSSH_3.7.1p2"`. This string match − normally very expensive in terms of processor time per packet − isn't bad at all if we've already figured out with the previous iptables command that this is almost certainly an ssh protocol string. By essentially requiring "SSH−" in the first 4 bytes of the connection, we avoid the problem of the string match incorrectly matching "SSH−" when someone downloads this article over port 80.

## Wrapup

With homage to Douglas Adams, Don't Panic.

The examples we've worked through have been annoyingly complex. The nice part? You don't have to rewrite them every time you want to inspect something else in the icmp header, or tcp payload, or anything else. Just grab the appropriate tests from above − or the table below − and finish them off with the specific field you wanted to inspect.

## Tests

First, a recap of the above, then some additional tests.

*"20"*
    Test for IPID's between 2 and 256

*"0FFF"*
    Check for packets with 256 or more bytes.

*"5*
    Match packets with a TTL of 3 or less.

*"16=0xE0000001"*
    Destination IP address is 224.0.0.1

*"1280F00"*
    Source IP is in the 192.168.15.X class C network.

*0x08*
    Is the TOS field 8 (Maximize Throughput)?

*"3>*
    Is the More Fragments flag set?

*"6*
    Is the packet a TCP packet?

*"4*
    Is the fragment offset 0? (If so, this is either an unfragmented packet or the first fragment).

*"0>>22de>*
    Is the TCP Sequence number 41? (This requires the previous two checks for TCP and First Fragment as well)

*"0>>221"*

Check for ICMP type=3 and code=1 (needs UDP and first fragment tests too)

`"0>>22"`

Is the UDP destination port 53? (Check for udp and first/no fragment first)

`"0>>221"`

Check that the UDP DNS Query bit is set (again, check for UDP, first/no fragment, and dest port 53 first).

`"0>>2212>>260=0x5353482D"`

Are the first 4 bytes of the TCP payload "SSH−"? (this requires a number of additional checks; the entire line is: `iptables −p tcp \! −f −m connbytes −−connbytes 0:255 −m state −−state ESTABLISHED −m length −−length 46:375 −m u32 −−u32 "0>>2212>>260=0x5353482D"`

And now, some new tests:

`"6`

Is this an ICMP packet? (From Don Cohen's documentation)

`"6`

Is this a UDP packet?

`"4`

Is the fragment offset 0 *and* MF cleared? (If so, this is an unfragmented packet).

`"4`

Is the fragment offset greater than 0 *or* MF set? (If so, this is a fragment).

`0>>22@−3>`

Is there any payload on this tcp packet (check for tcp and not fragmented first)? This elegant test was contributed by Don Cohen as I fumbled for a way to look for payload on a syn packet. By simply testing to see if payload byte 0 has a value between 0 and 255, we get true if payload byte 0 exists (read: "if there is any payload at all"), and false if we've gone beyond the end of the packet (read: "if there is no payload").

`"0>>22de>`

Is the TCP ack field 3000? (This, too, needs the TCP and First fragment checks)

`"0>>220"`

Is the high order bit (CWR) of the TCP flag byte (13) turned on? (Needs TCP and first fragment checks too). To see if it's off, use =0 at the end instead of =0x80.

`"0>>220"`

Is the next bit down (ECN−Echo) turned on? Likewise, =0 to see if it's off.

`"3>`

Is the More Fragments bit set, and are the Reserved and Don't fragment flags cleared?

`"3>`

This also checks if MF is set, but doesn't care about Reserved or Don't fragment.

`"3>`

Are More Fragments and Don't fragment set, and is Reserved cleared?

`"3>`

Is reserved set, and are More Fragments and Don't fragment cleared?

`"0>>22`

Is the ICMP ID (inside an echo request or reply packet) 0? This assumes you've already checked that the packet is icmp, unfragmented, and an echo request or reply.

`"0>>22"`

Is the ICMP echo Sequence number 35?

---

Don Cohen wrote the u32 module, and also wrote some (if you'll forgive me) somewhat cryptic documentation inside the source code for the module. William Stearns wrote this text, which borrows some examples and concepts from Don's documentation. Many thanks to Don for reviewing an early draft of this article. Thanks also to Gary Kessler and Sans for making the TCP/IP pocket reference guide freely available.

---

William is an Open−Source developer, enthusiast, writer, and advocate from New Hampshire, USA. His day job at SANS pays him to work on network security and Linux projects.

Last updated 12/18/2003.