Introducing Heyoka: DNS Tunneling 2.0

He was always running around with a hammer trying to flatten round and curvy things (soup bowls, eggs, wagon wheels, etc.), thus making them straight.

-- John Fire Lame Deer, Seeker of Visions

Alberto Revelli

r00t@northernfortress.net
ayr@portcullis-security.com

Nico Leidecker

nico@leidecker.info
nfl@portcullis-security.com

About us...

Alberto Revelli

- Senior Consultant at Portcullis Computer Security Ltd
- Technical Director of Italian Chapter of OWASP
- Co-author of the OWASP Testing Guide 2.0
- Developer of sqlninja - http://sqlninja.sourceforge.net

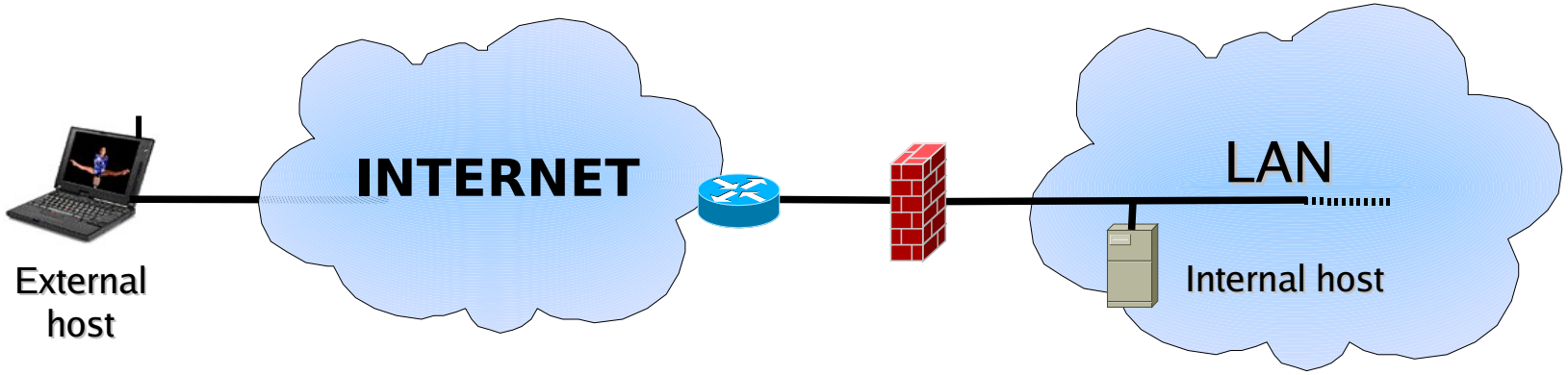
Nico Leidecker

- Consultant at Portcullis Computer Security Ltd
- Researcher on low-level protocol security
- Bug hunter

Agenda

- What is this all about
- Making the tunnel faster...
- … and also less detectable
- It seemed easy at first!
- Demo :)
- Key points and future improvements

So, why are we here anyway?



Goal 1. Make an internal host communicate with an external host, even if the firewall in between does not want us to.

Goal 2. A reasonably fast communication

Goal 3. Hide the channel to firewall/IDS/IPS

Possible scenarios and purposes

- **The Good:**

  You are a penetration tester and want to be able to set up a communication channel with a box you exploited

- **The Bad:**

  You are in an airport lounge and you want free Internet

- **The Ugly:**

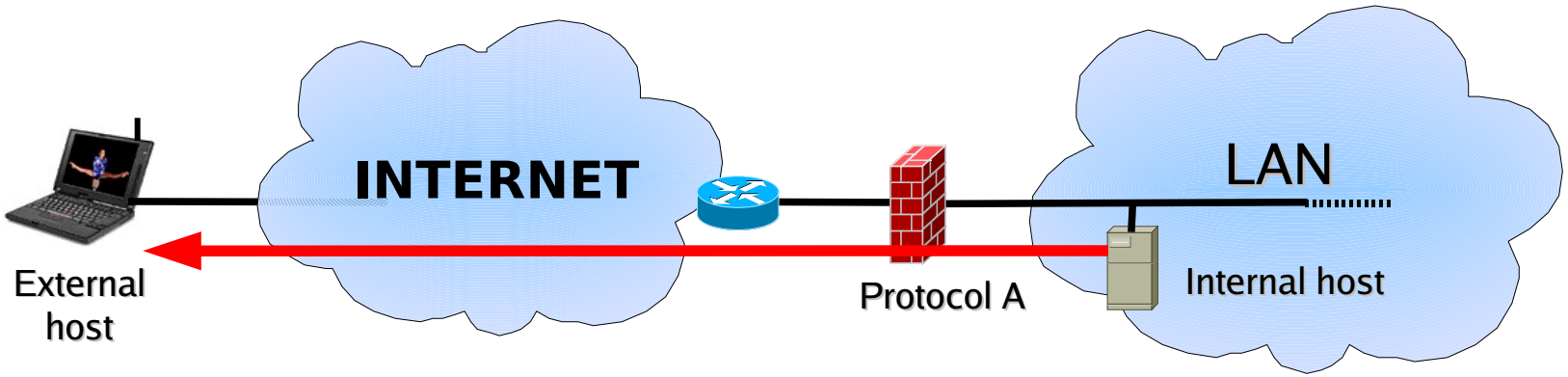
  You are a corporate spy and want to steal all .doc, .ppt and .xls files (plus all keylogs!) from all the boxes in your target network that just downloaded your custom trojan

The easy (aka: uninteresting) case

The most straightforward solution is to find a port that is allowed by the firewall, and use it to communicate to the outside

This allows us to contact the target machine (Goal 1), and the communication is likely to be as fast as we need (Goal 2), but such communication would be likely to be detected/logged by firewall and IDS (Goal 3)

An example is netcat, or SSH on a non-standard port

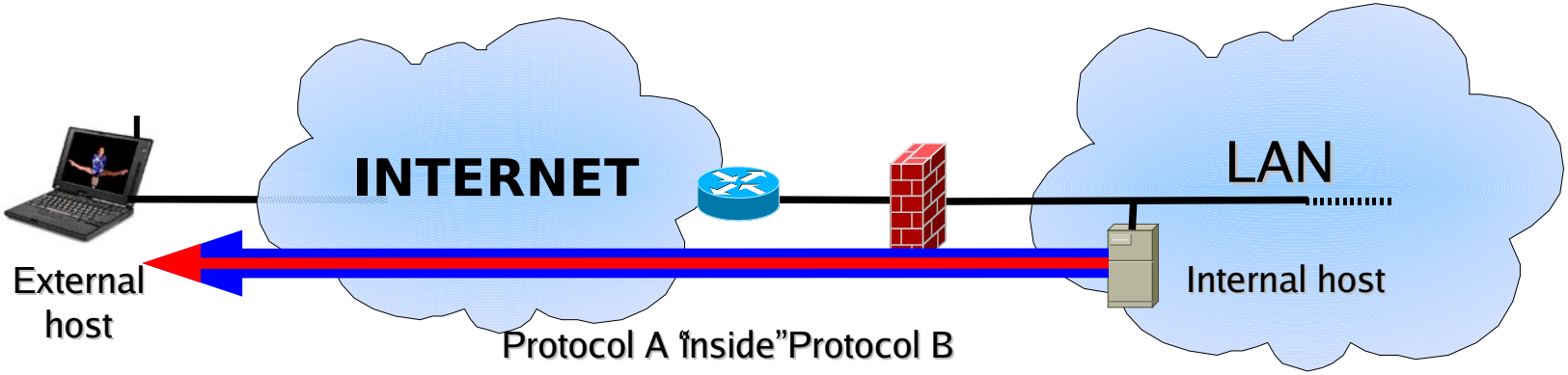


The not-so-easy case

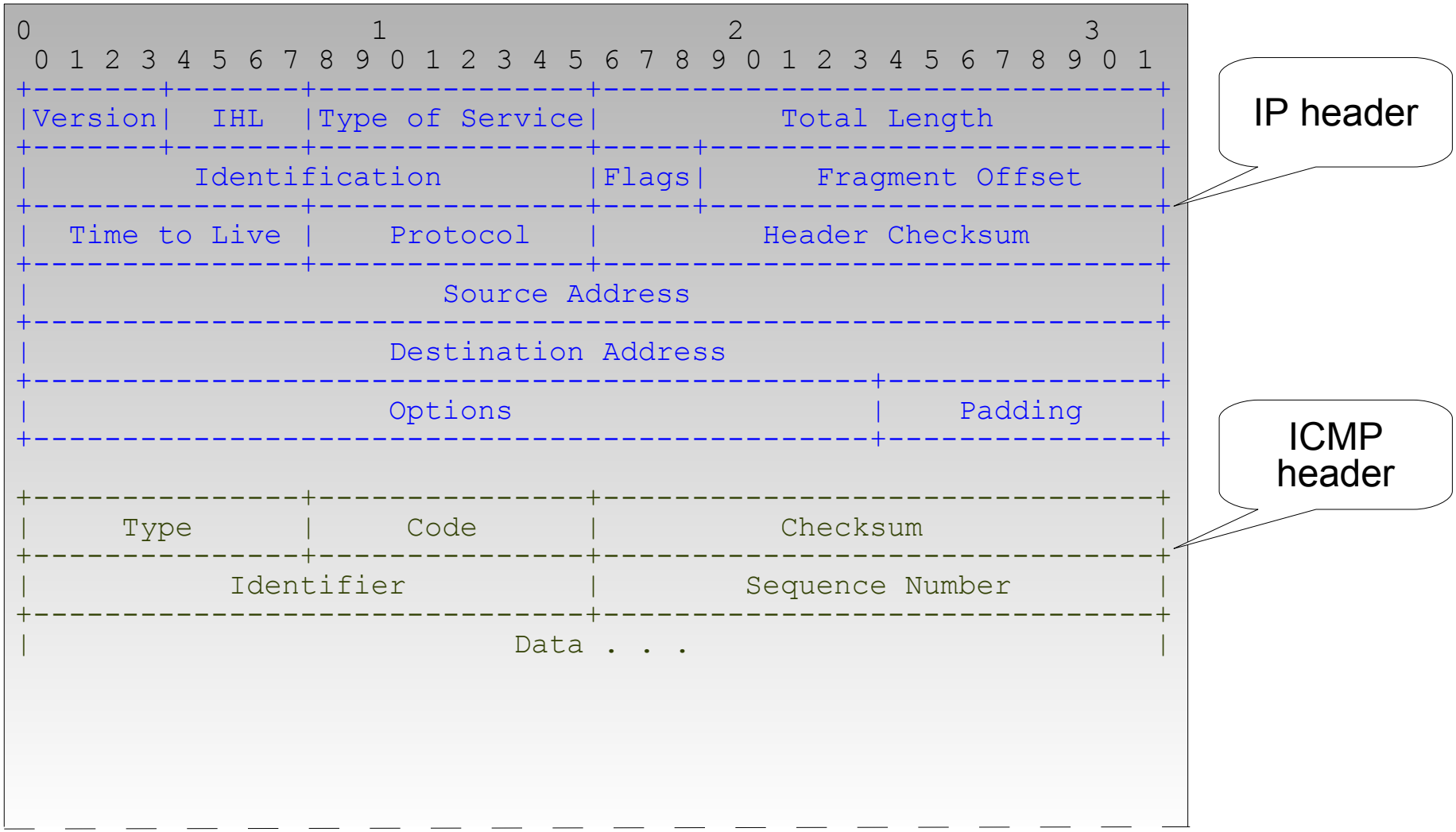
In other cases, all TCP/UDP ports are blocked but there is some other protocol that is allowed by the firewall

In this case, all we need to do is to encapsulate our protocol (e.g.: SSH) into the packets of the allowed protocol (e.g.: ICMP)

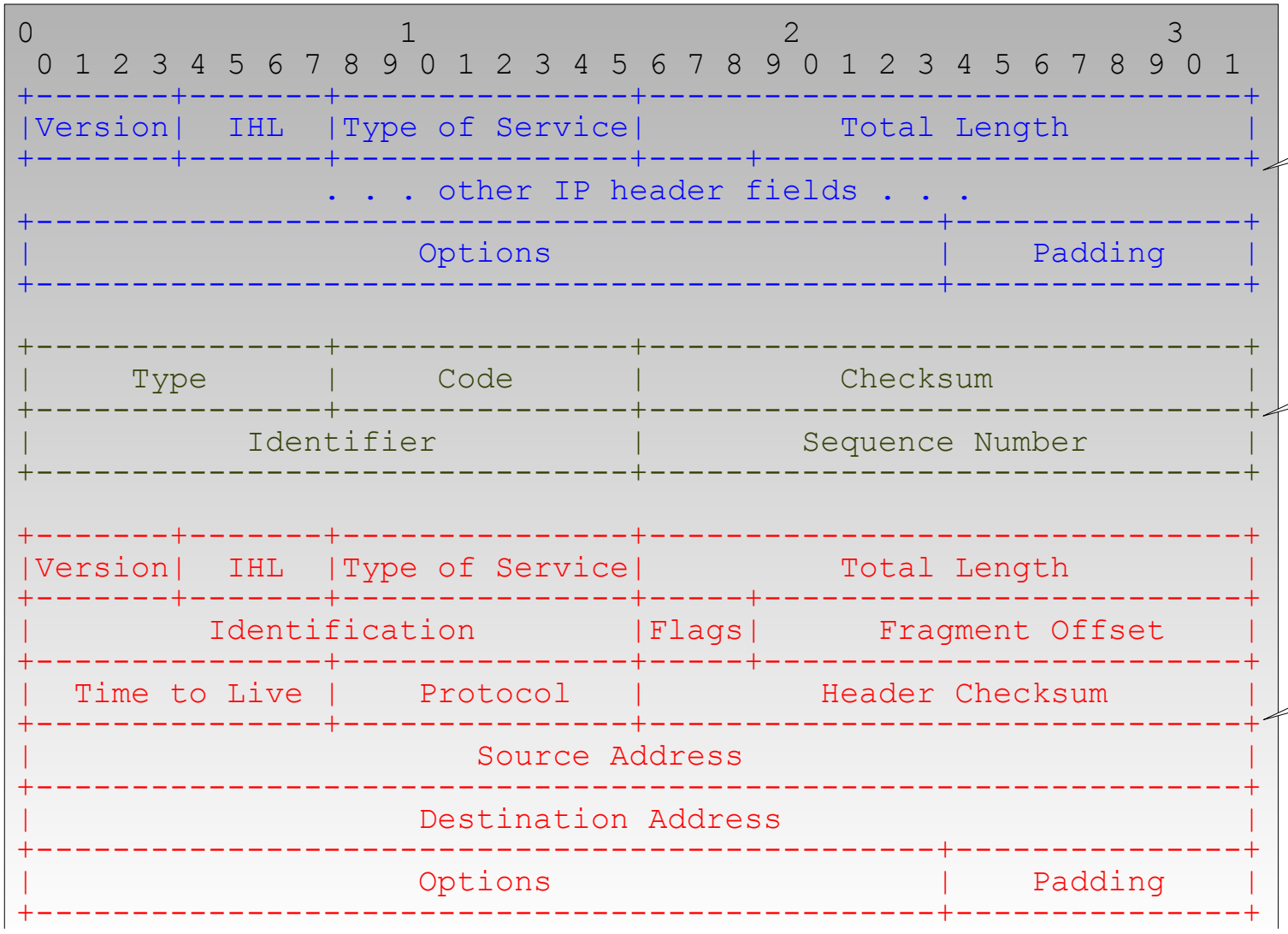
In this case, to detect this tunnel the payload of the transferred packets must be inspected. It is probably going to be reasonably fast too.



Example: tunneling over ICMP



Example: tunneling over ICMP (cont.)



IP header

ICMP header

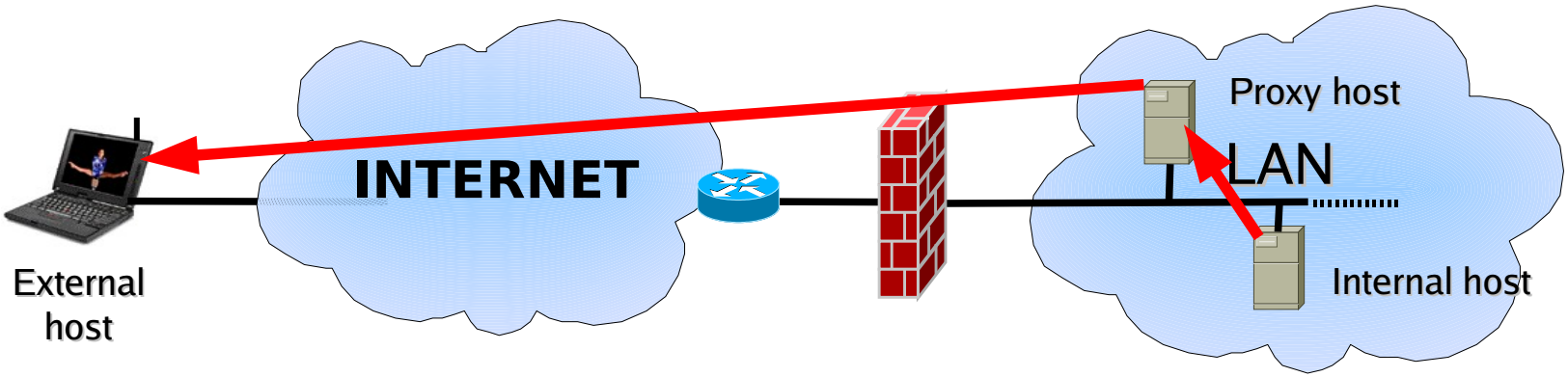
Tunneled IP header

The interesting case

The really interesting case is when there is **no** protocol that is allowed through the firewall, and no way to establish a communication between the two machines.

Or, more precisely, there is no way to establish a direct communication

The solution is to use a "proxy" machine that is allowed to establish a direct communication, and use it to piggy-back our traffic



A few options: HTTP Proxy

Pros

- Easy to implement
- Lots of networks have a few HTTP proxies allowed to access the Internet
- Just scan the internal network for open ports like 8080/tcp and 3128/tcp
- The proxy might be already configured on the "pwned" machine

Cons

- The proxy might be authenticated
- Even if authentication credentials are stored on the pwned box, they might be tricky to recover (e.g.: you might be accessing the Windows registry as a different user)
- Proxy requests are likely to be logged

A few options: e-mail

Pros

- Also easy to implement
- A SMTP server is quite likely to be allowed to send emails to any domain
- Just scan for an open port 25/tcp and try sending an email

Cons

- The server might be authenticated
- The communication would be quite slow
- The channel would be only half-duplex, unless you are able to read email sent to some user on the target network
- Also, mails are likely to be individually logged

Our favorite alternative: DNS!

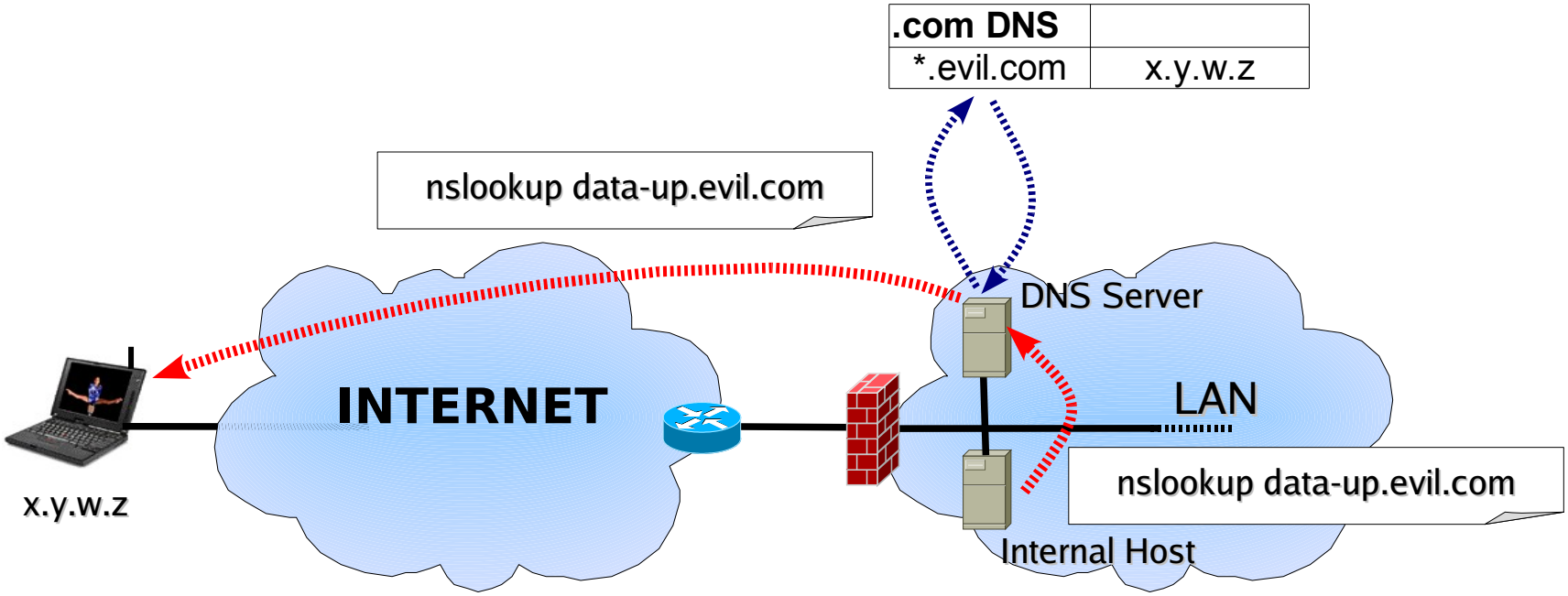
Pros

- Internal hosts are almost always authorized to resolve external names
- In our life as pen-testers, this happened in the vast majority of cases
- DNS requests are not likely to be logged
- Networked hosts have their DNS servers already configured. No need to scan for our proxy machines

Cons

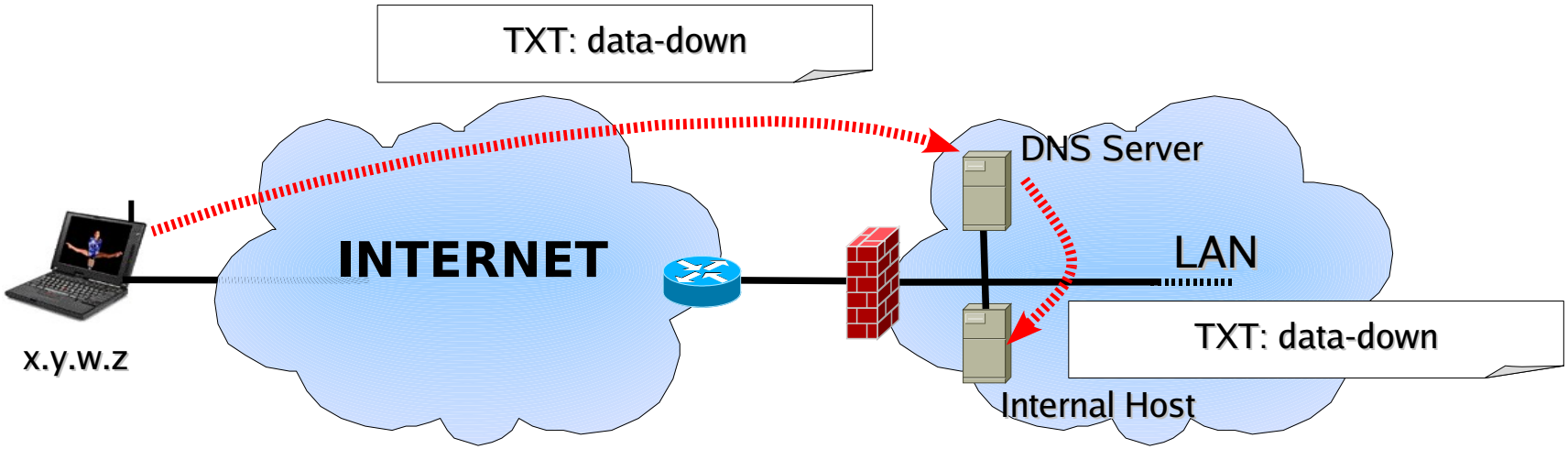
- You don't have much space in the packet, which makes the tunnel a bit slow
- You need to control the authoritative server for a domain, but this is less than ~20$/year with <insert domain registrar advertisement here>

A simple DNS tunnel (1)



- The host tries to resolve a hostname belonging to evil.com. The hostname contains the data to transmit
- The request is passed to the local DNS Server, which queries an external DNS server to obtain the IP address that is authoritative for evil.com
- The external server responds with the IP address of the attacker (x.y.w.z)
- The DNS server sends the request to x.y.w.z. The data has been received

A simple DNS tunnel (2)



- The attacker's host sends the DNS response, with the data to transmit back
- The amount of data transmitted back and its encoding depends on the type of the original query. In this case, for instance, a TXT query
- The response reaches the DNS server, and is forwarded to the internal host. Full duplex communication

Existing tools

So far, nothing new. The technique has been known for years and several tools exist to play with it, among which:

- OzymanDNS, by Dan Kaminsky - http://www.doxpara.com
- Iodine, by Bjorn Andersson and Erik Ekman - http://code.kryo.se/iodine/
- Dns2tcp, by Olivier Dembour - http://www.hsc.fr/ressources/outils/dns2tcp
- NSTX, by Florian Heinz and Julien Oster - http://savannah.nongnu.org/projects/nstx/
- Squeeza, by Marco Slaviero and Haroon Meer - http://www.sensepost.com
- ...And also sqlninja, even if just half-duplex for now (for the other half SQL injection is used instead) - http://sqlninja.sourceforge.net

Ok, now what?

The basic DNS tunnel technique that we have introduced suffers from two main drawbacks:

- DNS packets do not allow much data to be stored into them. This is not good if we want to transfer large amount of data (e.g.: all .doc and .xls files in the box) or we want a decent bandwidth (e.g.: for a VNC/RDP connection)
- Setting up a DNS tunnel to transfer more than a few bits generates a large amount of traffic. Detecting and locating a DNS tunnel is easy, when you spot a single IP address that generates half of your DNS traffic

What we did was to have some fun trying to limit, when possible, the impact of both these problems

Agenda

- What is this all about
- Making the tunnel faster...
- … and also less detectable
- It seemed easy at first!
- Demo :)
- Key points and future improvements

Hostnames do not provide much space

We know that, upstream, in order to store our data we can only rely on the hostname, whose structure is described in RFC 1034:

- It can be up to 253 characters
- It is structured in different labels, each of which can be up to 63 characters long
- Each character can be a letter (case insensitive), a number or an hyphen

This means that if we are using the domain evil.com the largest possible hostname will be structured as follows:

```
[63chars].[63chars].[63chars].[52chars].evil.com
```

We therefore have 241 characters to play with, each of which can assume 37 different values.

Even if RFCs state that one request can contain multiple hostnames, actual implementations do not allow that

So... how much data?

In order to encode our data, we can therefore use base32, storing 5 bits per character.

In our example, we would be able to send 241x5 bits per packet

We need to allocate some space to ensure reliable communication (e.g.: packet counter)

We end up with ~150 bytes per upstream packet

Even gzip'ing everything before transmission, it's not a lot :/

Example of base32 encoding

```
Standard query A 7brbj6gnczqynpn2pk7nwuqauo5777mrk6pm5fqegwbtb3udameqzmpl3gnd.mo2ibhszs3atk3q2xcwt3cijoh6g4gx7l4bje2nwc
Standard query A aaaaadakcuaaaaaaaaaaaaaaaabruubbdnkslmenc5gkmmppg2llaf3rzw7nr.dmqvrzw7q2blobhca7cycclypbgjypqwwaz67igbw
Standard query A aaaaafageiaaabaaaaaaiaaaaaqaaaaaaaaaaaa.17692-0.id-17316.up.sshdns.sqlninja.net
Standard query A aaaabdafeaaaaaebac4rme4gvi6dzvfsfjwvhzbxeh76ncgciprzhteltnkl.6ua5ble5gyzo3mzbkc4oqakpnowi2rq4cwlnjuiwk
Standard query A amj5e2tnqbqnqdgpvngvujat6deq7qw7wrvs3cab63l47dl4kwjq.52227-0.id-17316.up.sshdns.sqlninja.net
Standard query A auwopjup3xbg4t3folpp6lmnhe43xcjxnqi66p2s2ribpkpa2xga.15588-0.id-17316.up.sshdns.sqlninja.net
Standard query A bmlrx3f4o4tueiam5jjfjnxl6xnnztyvdacdfcby5azmzayaaaaaaaa.24008-0.id-17316.up.sshdns.sqlninja.net
Standard query A bssicc4kgvt5eonqyx5sj4m42ksybxeomsjvo2lpyleumdyvw7bq.52828-0.id-17316.up.sshdns.sqlninja.net
Standard query A en2elgm534at7ev3s7fmm73p5ybtwwbqzioioznf7f3oldmcdpla.10928-0.id-17316.up.sshdns.sqlninja.net
Standard query A fvrwey2anr4xgylun5zc43djouxhgzjmmfsxgmjshawwg5dsfrqwk4zrhezc.2y3uoiwgczltgi2tmlldorzaaaaakvug2yldfvwwi
Standard query A gaax2sormid3kmhsz5wrsr2wx6ifvlvjzhd7tox66okxnz6agqmq.48928-0.id-17316.up.sshdns.sqlninja.net
Standard query A geg5zf4bwh6c336e3dmhmxycqgdm7gzuzsifou6fkguv2ouqsotq.10480-0.id-17316.up.sshdns.sqlninja.net
Standard query A j4ogghsrcxzx6kh4anyzl5hoegmca6ulxfjktr5zbwxdoi6skpfm4kdfnrnj.mnbzmsf5culqb2trs.47594-0.id-17316.up.ssh
Standard query A j4oxewjikbzwbvdlwycvz43g673e6sczv7yq4jnux6ier5xfmmacd2hmn3qu.nurunh5brzjfwbukibwesojj2n3jqqudo5kxiq2b4
Standard query A jtocpnxgrcvl5rnlf4pdpxds4s3d237f5q6p7sveendedgwqb6ma.11757-0.id-17316.up.sshdns.sqlninja.net
Standard query A knjuqljsfyyc2t3qmvxfgu2il42c4m3qgiqeizlcnfqw4ljzmv2gg2btbiaa.aaweaukk2li3p3nm2ncpq4jnecljy6cvwaaaabmwi
Standard query A l7eolg6amqjs6lj5bkuuoprcj736e2ckh6abfxdaziudyp7ccivq.54804-0.id-17316.up.sshdns.sqlninja.net
```

Here's an example of data transferred with OzymanDNS

Are we really limited to base32?

So far so good, but reading some other RFCs we discover interesting things:

- "[..] any binary string whatever can be used as the label of any resource record"- RFC 2181
- "[...] the individual octets of which DNS names consist are not limited to valid ASCII character codes. They are 8-bit bytes, and all values are allowed"- RFC 4343

We might have more freedom when constructing the hostname that contains our data

However, gethostbyname() and getaddrinfo() require a NULL-terminated string as the hostname, so we cannot use them to send binary data.

Also, how does a DNS server handles the hostname length, if it might contain binary data (which might contain NULL bytes)?

How a hostname *really* looks like

If you run wireshark and capture the previous DNS request for evil.com, here's what you'll see in the hostname field:

```
[0x3F][63chars][0x3F][63chars][0x3F][63chars][0x34]
[52chars][0x04]evil[0x03]com[0x00]
```

Each label is prepended with its length, so NULL bytes might not be a problem, after all

We coded up a small C program that forges a DNS request with binary data, tried through BIND9 and guess what? It worked fine!

This means 8 bits per characters, instead of 5

...Or, if you prefer, a tunnel that is **60% faster**!

How servers support binary hostnames

DNS Server	Operating System	Result
BIND9	Linux/UNIX	OK
djbdns	Linux	OK
MS DNS Server	Windows 2003 Server	The request is forwarded, but the response triggers a 'server failure' message

It all looks quite encouraging, and for Windows we will see that the server failure can be handled

What about downstream data?

- The response can already use binary data, by using NULL queries, defined in RFC 1035, and tagged as "EXPERIMENTAL" since then (Iodine uses this kind of request)

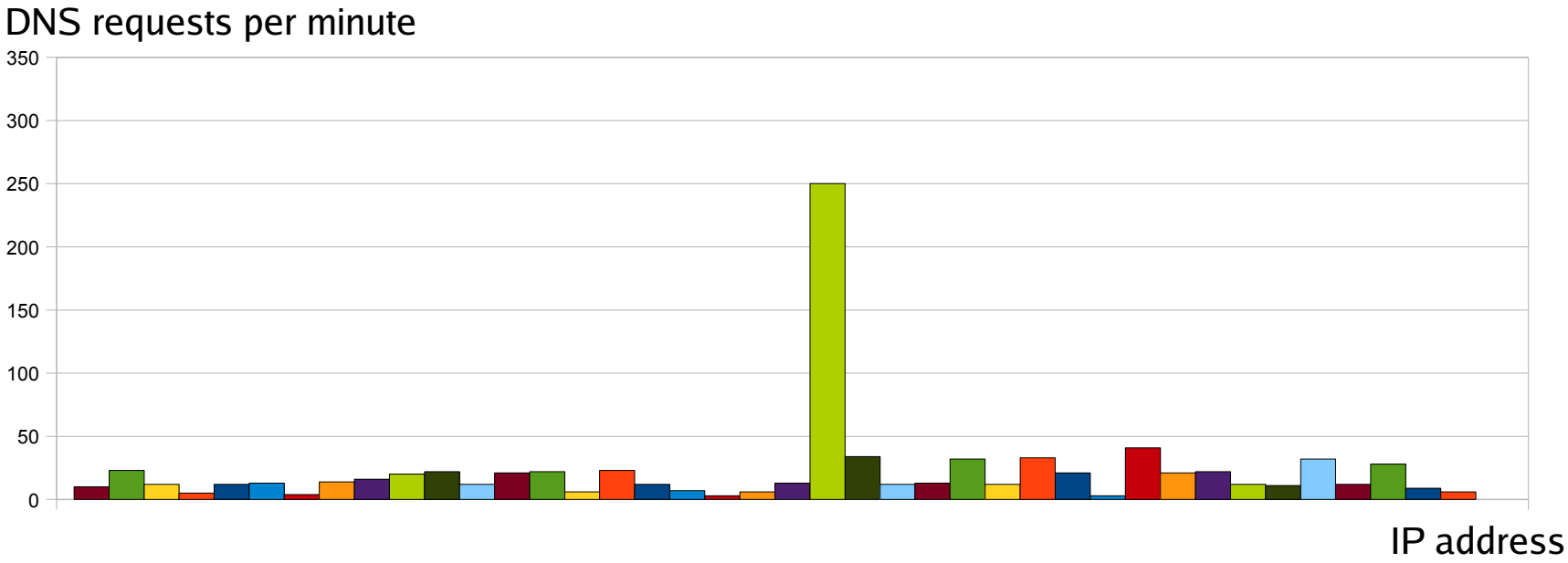
- In case NULL queries are not allowed or not supported, binary encoding seems to work also in TXT responses. In this case, since we move from 6 bits per character to 8 bits per character, the bandwidth is increased by **33%**

Agenda

- What is this all about
- Making the tunnel... faster
- … and also less detectable
- It seemed easy at first!
- Demo :)
- Key points and future improvements

The detection problem

As mentioned before, one of the problems in setting up a DNS tunnel is the high amount of DNS traffic generated by the tunnel endpoint:



An anomaly-based IDS is very likely to notice such a traffic pattern

“Spreading” the tunnel

A spoofed DNS request is quite likely to reach its destination and deliver the encoded data: since DNS uses UDP we don't have to worry about guessing TCP sequence numbers.

Spoofing the source address, using each time a different IP address belonging to the same network, would “spread”the tunnel signature among all hosts, making the data transfer a little harder to detect

Switches now offer port-level anti-spoofing protections, but in our experience only a few administrators enable them

“Spreading” the tunnel (cont.)

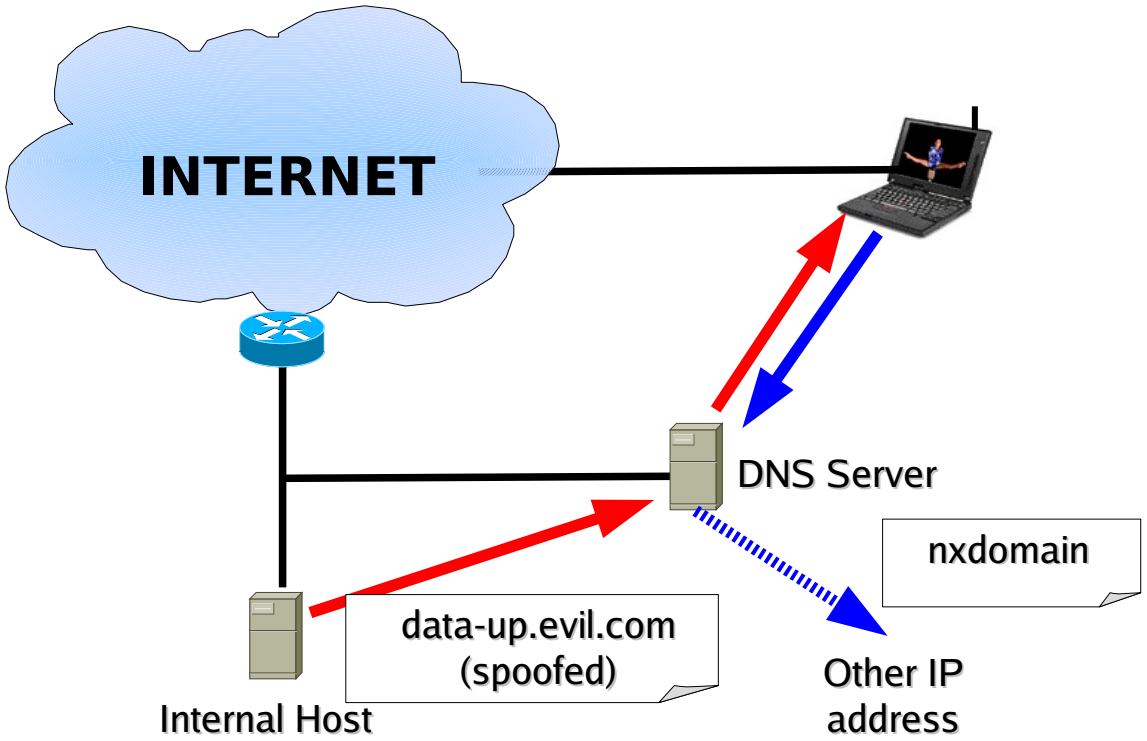
We are already building our own packets to transfer binary-encoded data, so it's not a big deal to mess up with them a little more

However, the DNS responses would obviously be sent to the spoofed source IP address, and therefore will never reach our host. So we would be limited to a half-duplex channel

...or not?

Splitting the tunnel in two

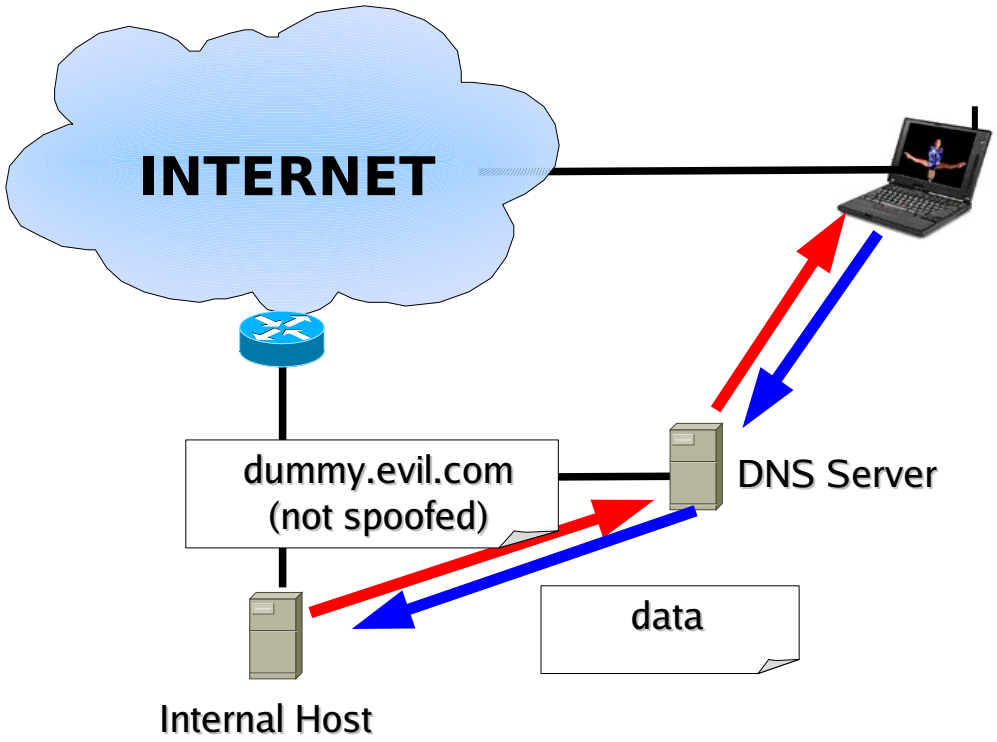
We create 2 different, concurrent communication channels: one uses spoofed packets to transfer upstream data:



All upstream data is trasmitted using spoofed packets. The response will be sent to the spoofed IP address, so we can simply respond with NXDOMAIN

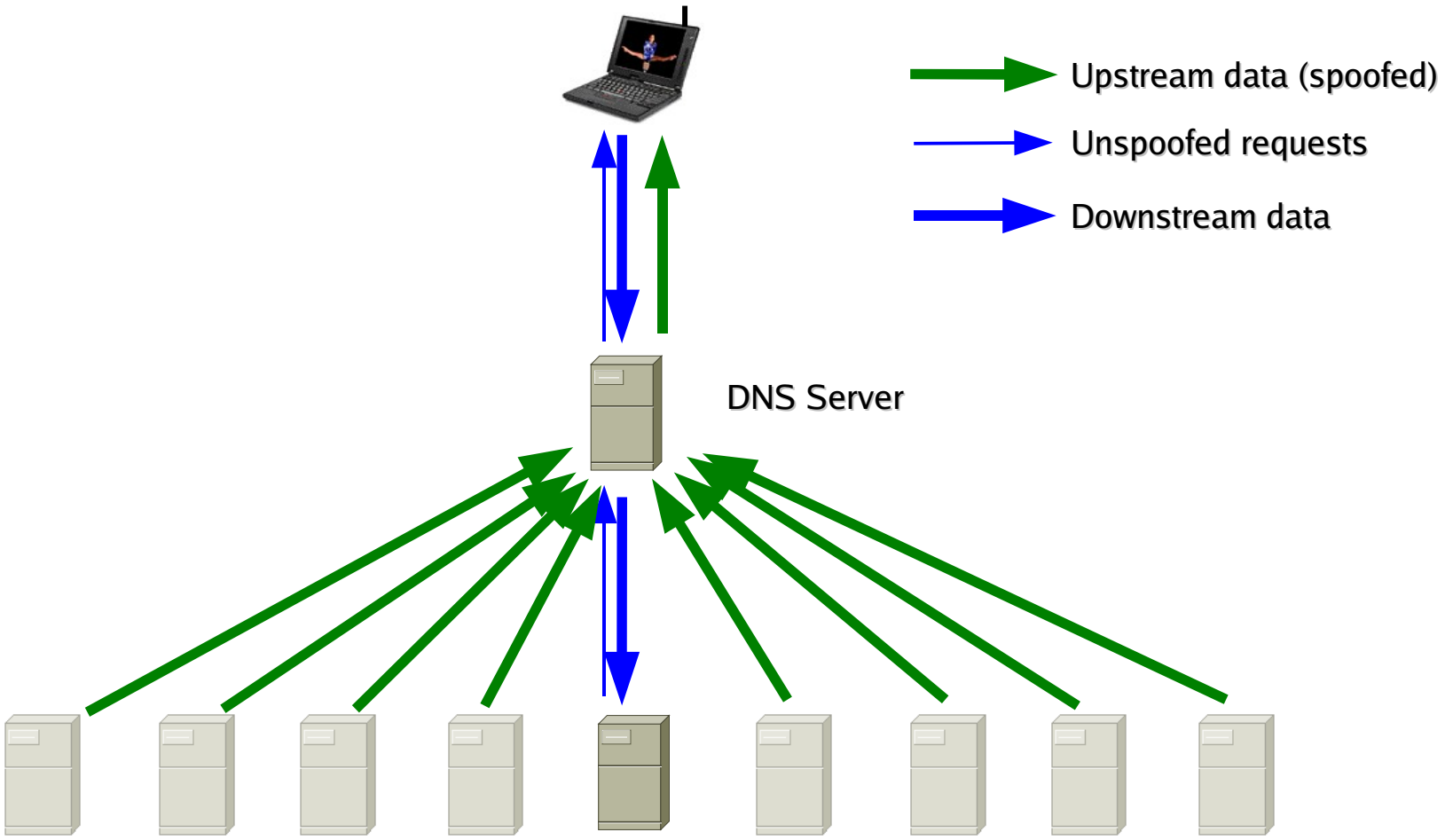
Splitting the tunnel in two (cont.)

The second tunnel sends out unspoofed requests (without data encoded), and waits for the responses, which will contain the downstream data



- Unspoofed queries can be used not only to receive data but also as a control protocol (heartbeat)
- The fact that unspoofed queries do not need to send data solves the issue with the Windows 'server failure' messages

How the tunnel looks like



A bit of analysis...

- By splitting the channel, we overall need more packets for the same amount of data
- If the channel is perfectly symmetric (same amount of data per packet in each direction, same amount of overall data to transfer in each direction), we actually double the amount of traffic
- The only advantage would be that those weird requests with very long hostnames encoded in binary would be spoofed. This would make the investigation on the actual tunnel endpoint a little harder

However, DNS tunnels tend to be very asymmetric!

...On the beauty of asymmetry

- Using EDNS0 (defined in RFC 2671 to allow DNS messages larger than 512 bytes over UDP), TXT responses can store up to 1024 bytes
- This means that, in a fully symmetric data transfer, we might only need 1 response every ~4 requests
- ...But wait! There is more!

…On the beauty of asymmetry (cont.)

- If we want to tunnel a RDP session, we will have a lot more upstream data (the graphics) than downstream (keystrokes and mouse movements). In a standard RDP session, upstream data can easily be 6-7 times the downstream data

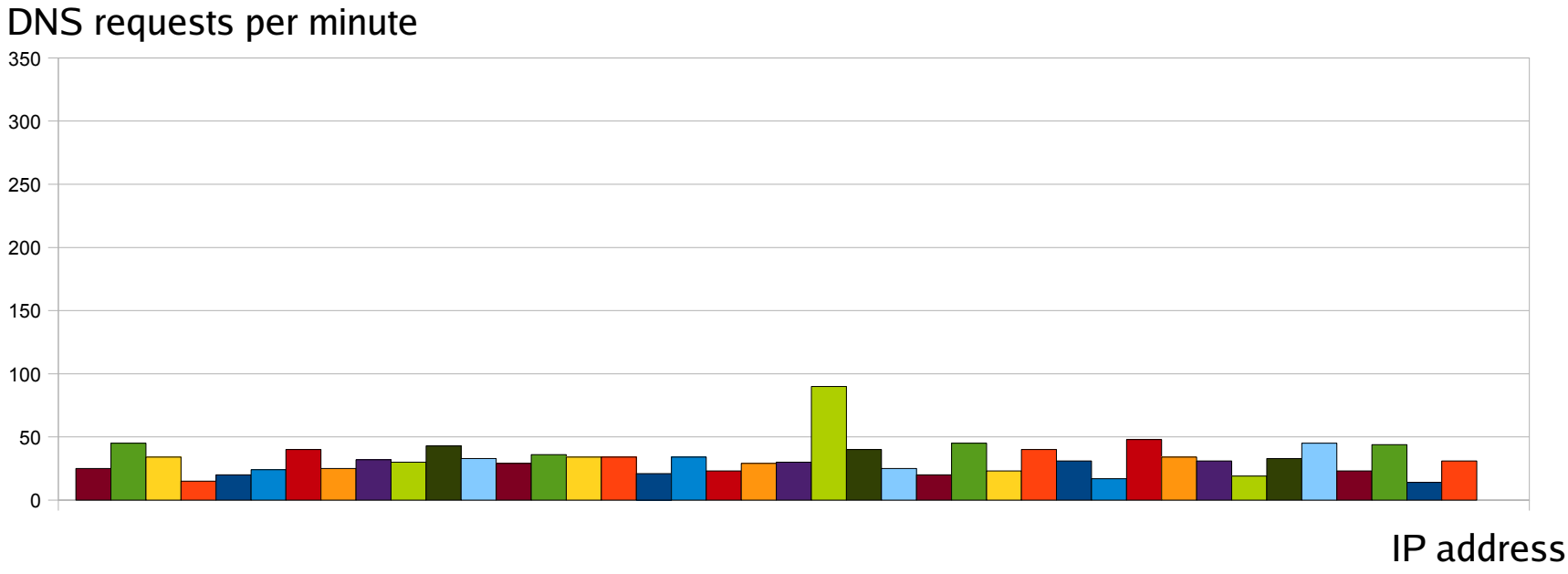
- Things are even better in the "corporate spy" case, where **all** data is upstream. The only downstream data would be used to communicate when a file has been successfully received (e.g.: by sending its SHA hash)

Wow, splitting&spoofing might actually be quite convenient!!

New traffic signature

Spoofing the IP addresses, the traffic signature is spread across the entire network segment.

Depending on the amount of downstream data and the number of spoofed addresses, the “real”IP address might still produce more traffic, but the difference will be less likely to be spotted right away



Of course, if we are aiming to only exfiltrate data, the peak will disappear :)

NXDOMAIN.... or not?

Sending a NXDOMAIN to spoofed requests is not strictly necessary. However:

- NXDOMAIN responses ensure that a stateful firewall never sees a suspicious number of unanswered queries
- While they increase the number of responses sent to the target network, they ensure that the internal DNS server does not re-send spoofed requests, therefore decreasing the overall number of packets

Agenda

- What is this all about
- Making the tunnel... faster
- … and also less detectable
- It seemed easy at first!
- Demo :)
- Key points and future improvements

Turning ideas into code...

It seemed to work fine, so we decided to quickly put together a tool to implement the idea.

Considering the weird packets that such a tool would generate, we decided to name it "Heyoka", a sacred clown in the native american tradition.

From Wikipedia: "Heyoka are thought of as being backwards-forwards, upside-down, or contrary in nature. This spirit is often manifest by doing things backwards or unconventionally -- riding a horse backwards, wearing clothes inside-out, or speaking in a backwards language."

Heyoka is currently in an extremely-early-beta stage. Thing is, it was not as simple as we thought at the beginning

Problem #1: packet loss

UDP does not guarantee packet delivery, and we are trying to tunnel TCP data

Functions like gethostbyname() and getaddrinfo() provide some kind of retransmission feature, but in our case we have to re-build everything from scratch

- We need some way to keep trace of packets
- Detect when a packet was lost
- Non spoofed requests/responses, beside providing downstream data transfer, can be used to tell both sides which packets have been lost and need to be retransmitted

Packet loss & retransmission

- How about a counter for sent packets and another for packets successfully received?
- Both endpoints need to locally cache sent packets, in case they need to be retransmitted. Packets can be flushed from cache as soon as the other side acknowledges them
- That might work, but remember that in our packets we are very short in space
- Yes, we are pretty much re-implementing an ultra-squeezed TCP :)

Payload layout

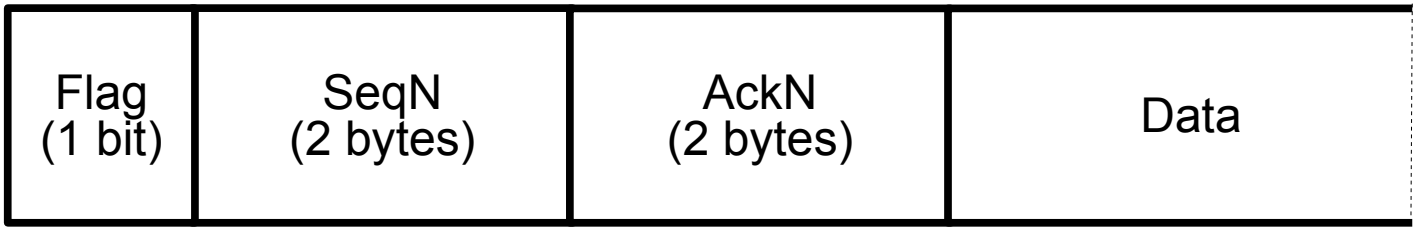
Problem #2: spoofed? Non spoofed?

Responses to spoofed requests get lost. Responses to unspoofed requests reach the other end (the "slave"), and are used for downstream data transfer

This means that the "master" needs to be able to detect when a request is spoofed and when it is not, in order to decide whether to include data in the response

However, from the master's perspective all requests appear to come from the same IP address (the address of the DNS Server), so we must use some other way to signal this information

- Seems that we need a flag, to tell when a request is spoofed and when it's not



- Spoofed (y/n)

Problem #3: Encoding

Are we really sure that binary data will successfully make it through every possible DNS Server?

RFCs are not always followed, or some IPS could drop DNS requests with binary hostnames

Unless we can contact the network administrator and ask him/her to let our packets through, we need some sort of "fall-back"feature, to use the good ol' base32/base64 encoding, when needed.

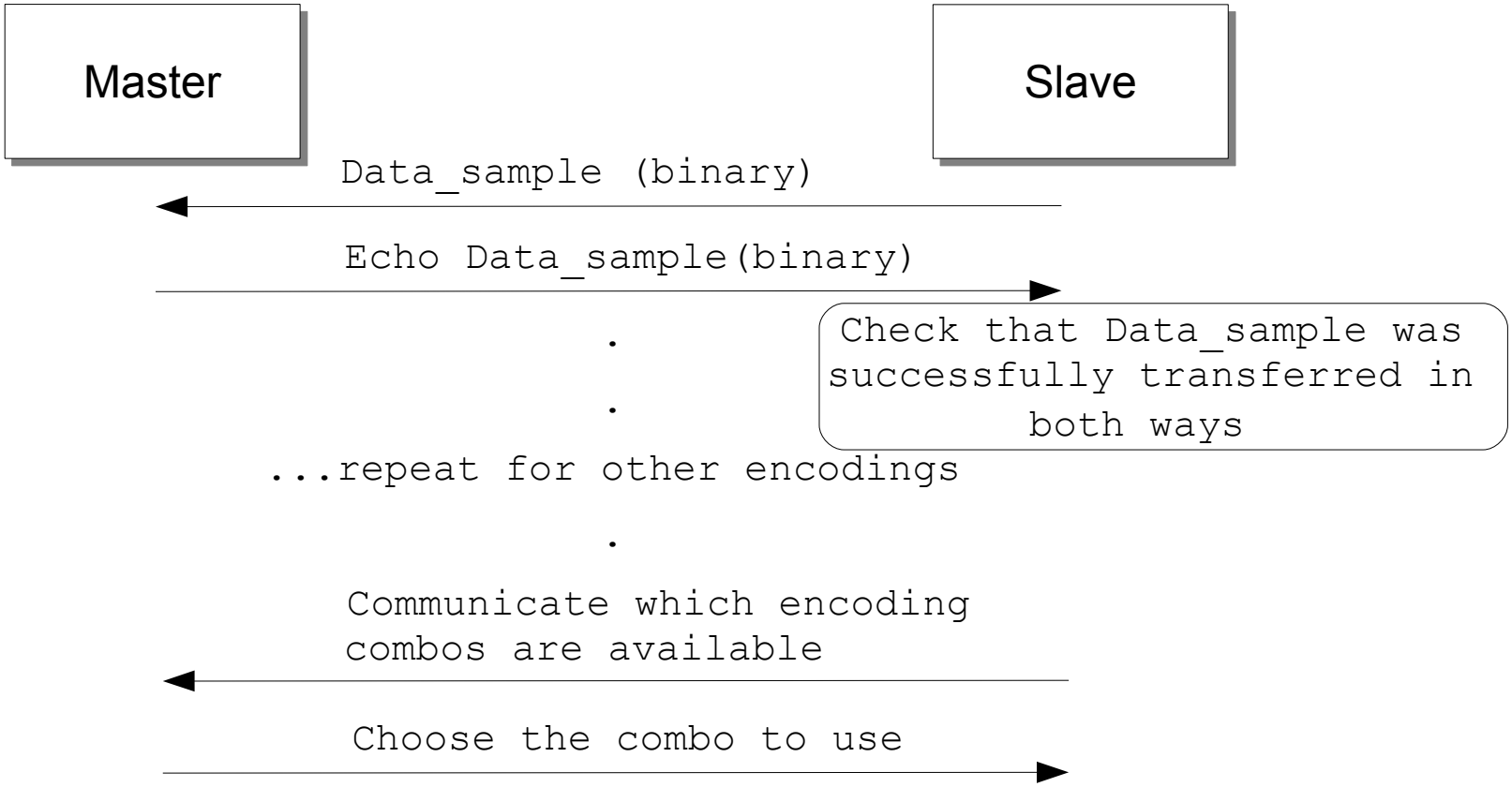
Problem #3: Encoding (cont.)

We need to be able to handle 4 different combinations that result from:

- 2 different encodings from the slave to the master (binary and base32)
- 2 different encodings from the master to the slave (binary and base64)

We need a "handshake" protocol to negotiate the best available encoding combination and then start the communication

A Handshake Protocol

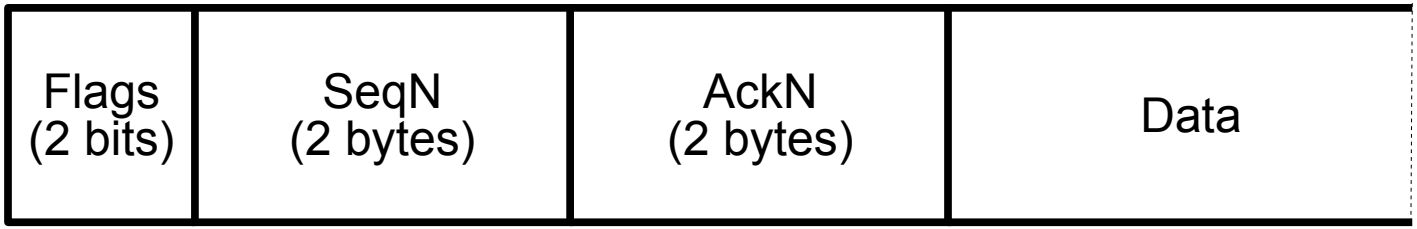


If stealth is more important than speed, one will choose a standard encoding, possibly using shorter hostnames too

Problem #4: Signaling the Handshake

This leads us to a new problem: the master needs to know whether a packet contains data or whether it is part of the handshake

- We therefore need to add a second flag:



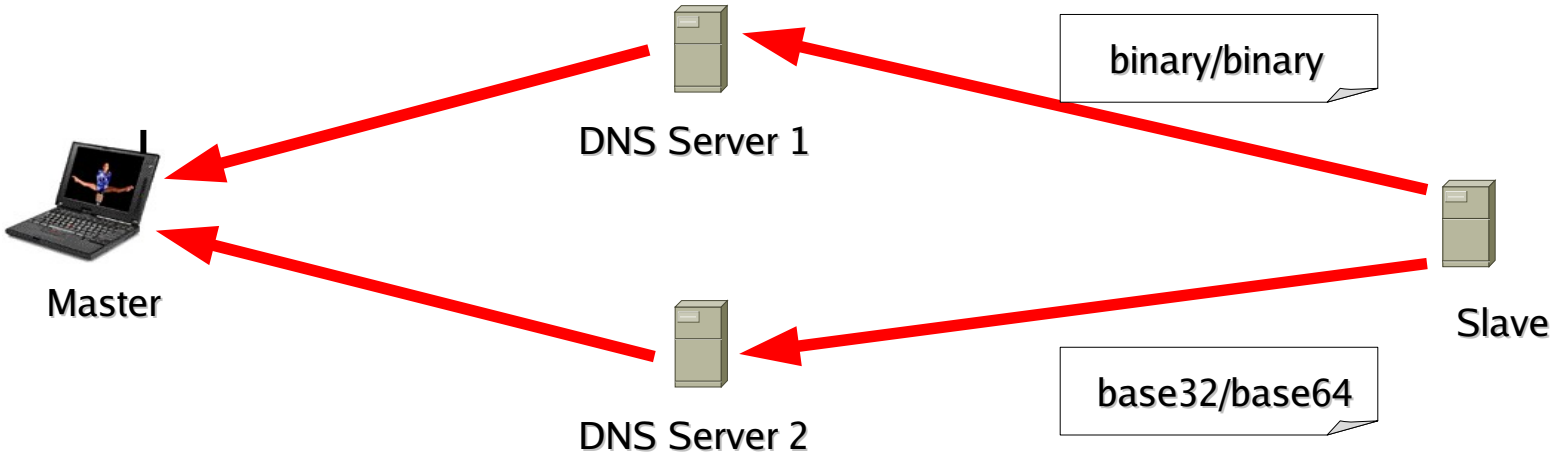
- Spoofed (y/n)
- Handshake(y/n)

Problem #5: Signaling the encoding

The master knows how a slave encodes data, since this was agreed upon during the handshake

However, a slave could run on a machine with multiple DNS servers configured, and those servers could support different encodings

We could just choose the one with the best encoding, but...



Multiple servers, multiple encodings

Instead of choosing one, we want to use all of them to tunnel our traffic

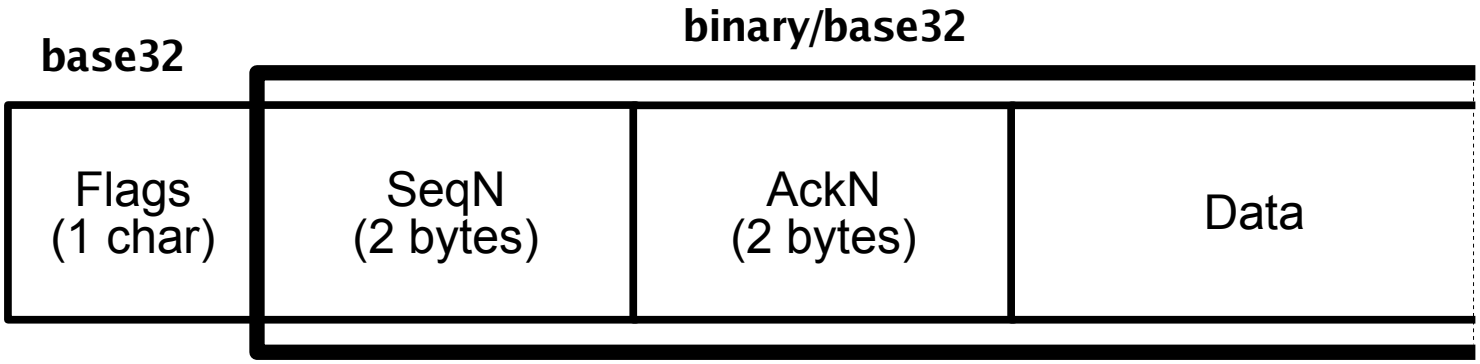
Being able to use multiple servers at once gives us some significant advantages:

- The tunnel is harder to detect
- We have more bandwidth
- If a DNS Server fails, the tunnel survives

However, using multiple servers with potential multiple encodings forces the slave to communicate to the master how each packet is encoded

Mhh... seems that we need to signal a few more things in that flag!

The new packet structure



Since the flags contain information about the encoding of the following data, they need to be "outside" of the encoding itself. Therefore, we use a first character that is **always** base-32 encoded (therefore storing 5 bits).

So far, the flags are the following:

- Packet spoofed / not spoofed (1 bit)
- Handshake / Data (1 bit)
- Slave->Master encoding (1 bit)
- Master->Slave encoding (1 bit)

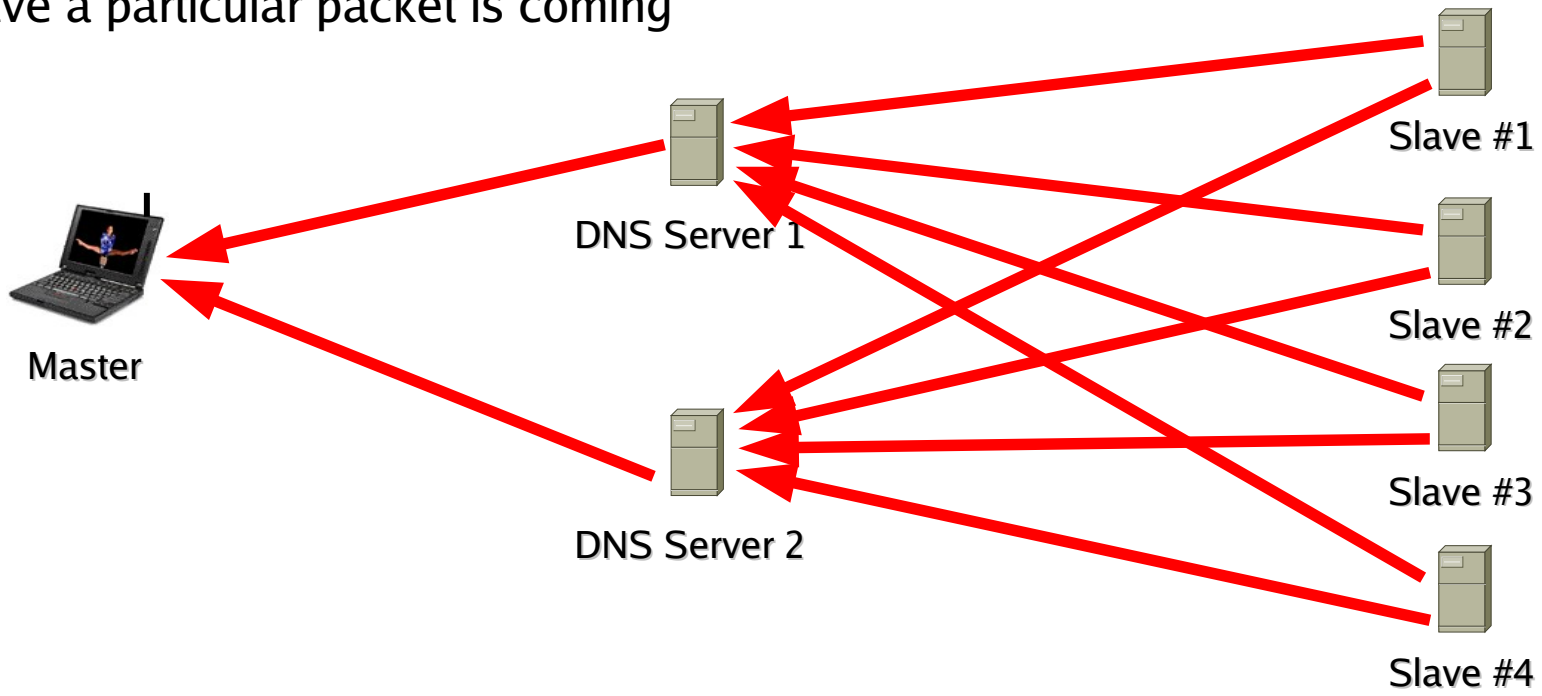
And we still have 1 bit left!

Problem #6: Handling multiple slaves

As a penetration tester you might be happy with one slave calling you from the target network but there might be cases in which you deployed multiple slaves on multiple machines.

This is especially true in the case of a trojan stealing files from infected machines.

Unless you want to use multiple domains, you need to be able to tell from which slave a particular packet is coming

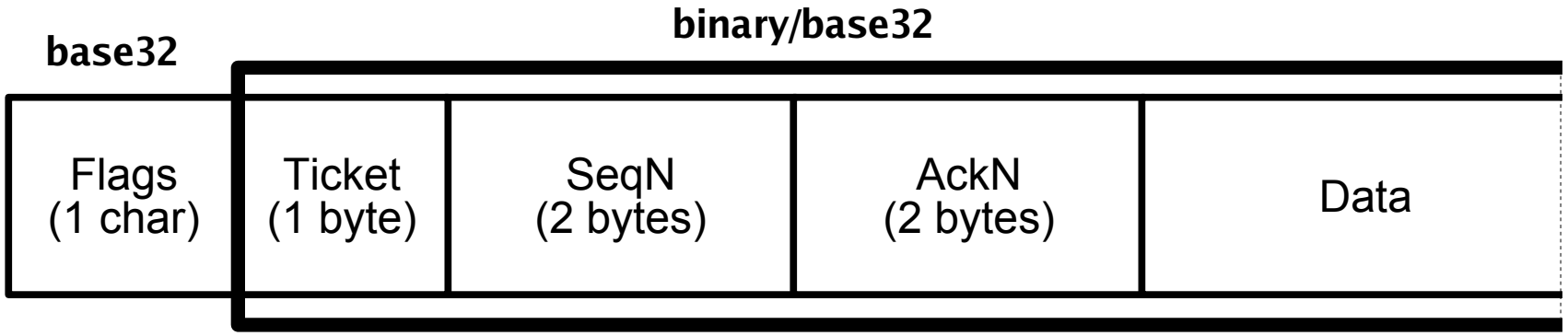


Handling multiple slaves (cont.)

One could use a different subdomain for each slave, crafting requests like the following: encoded-data.**n**.evil.com, where **n** is the slave identifier

A more efficient option is to allocate one last byte in the data part of the domain name (we call it "ticket")

Since the slave does not know how to generate a unique name by itself, the master can assign the ticket value at the end of the handshake



That should work. A decent control protocol in just 6 bytes!

Master-side signaling

So far we have seen the signaling that is performed from the slave to the master. In the other direction, luckily, things are a bit easier:

- Since the encoding is at least base64, we could use 6 bits in the flag, instead of just 5
- We just re-used the 5-bits base32 codec of the slave (we are lazy!)
- Encoding flags would not be strictly required, since the encoding is specified in the request. However, this makes the slave's work easier, since it uses two independent threads for reading and writing

Master-side signaling (cont.)

- The master needs to receive a DNS request from the slave, in order to transmit data. How often should the slave send such requests?
- Sending more requests (even when we don't have data to send) allows the master to send its data quicker, but generates more DNS noise
- We need an adaptive system: the master must be able to tell the slave when to send more requests (as there is data to send) and when to slow down (the master data buffer is empty)
- The master flags therefore include a “more data waiting”bit

A slave with adaptive timing

1) We define a timeout for sending non-spoofed packets (heartbeat) - e.g.: 500ms

2) If the last received response has the more_data flag set, we immediately send a new unspoofed request, in order to receive the data that is waiting

3) Otherwise, we check if we have data to send. If so, we send out a spoofed request

4) If no data is available, we wait for the timeout to pass before sending out a new heartbeat

Agenda

- What is this all about
- Making the tunnel... faster
- … and also less detectable
- It seemed easy at first!
- Demo :)
- Key points and future improvements

A possible exploitation scenario

- You have found a SQL Injection on a web application, with Microsoft SQL Server as backend
- You have 'sa' privileges, or you have been able to escalate privileges (e.g. using the heap overflow in sp_replwritetovarbin)
- Re-enable xp_cmdshell, using sp_configure
- Enable RDP, if necessary
- Add a user (e.g.: "heyoka")
- Upload heyoka.exe
- Run it!

Uploading/executing programs

Even if we can't establish a connection to the target server, it is easy to upload files in the described scenario

Any executable up to 64k can be “represented”as a debug.exe script, and debug.exe is shipped by default with all Windows versions

We can translate heyoka.exe to a debug script, write that script on the hard disk of the target server using xp_cmdshell, and then call debug.exe on it

Alternatively, a VB script can do the job as well

As for adding the user and enabling RDP, it's easy with xp_cmdshell and xp_regwrite

Agenda

- What is this all about
- Making the tunnel... faster
- … and also less detectable
- It seemed easy at first!
- Demo :)
- Key points and future improvements

Future improvements

- DNSSEC...?
- Abusing caches
- Code refactoring
- Porting to other platforms
- Some decent user interface + API
- Official release!

A few takeaways

- Forbidding your hosts to establish connections to the Internet does **not** mean that they are not communicating with the external world, right now
- A determined attacker can find lots of ways to exfiltrate data from a network
- Such data transfer can use spoofed traffic, making detection/reaction even more complex
- We used DNS, but other protocols can be used as well

Even more takeaways!

- In a nutshell, protecting against data exfiltration is **a hard job** (unless you want to get rid of TCP/IP and use unidirectional networks)
- In the meantime, do not allow internal hosts to resolve external names: use an authenticated proxy for all communications
- Modern switches provide port-level anti-spoofing protection. Use it!

Credits

Peer review, ideas, suggestions:

- Dan Kaminsky
- Christien Rioux
- Ollie Whitehouse
- Haroon Meer
- Angelo dell'Aera
- Stefano di Paola
- Oreste Bergamaschi
- Kieran Combes

SVN Server:

- Bernardo Damele A.G.

Moral support:

- C_2H_5OH

~~Do not~~ try this at home!

http://heyoka.sourceforge.net

r00t@northernfortress.net

nico@leidecker.info