# Staring Into The Abyss

Revisiting Browser v. Middleware Attacks In The Era Of Deep Packet Inspection

Dan Kaminsky, Director of Penetration Testing, IOActive

**PREPAPER EDITION**
**Please mail thoughts and corrections to:**
**dan@doxpara.com**

Summary:

DPI -- Deep Packet Inspection -- technology is driving large amounts of intelligence into the infrastructure, parsing more and more context from data flows going past. Though this work may be necessary to support important business and even security requirements, we know from the history of security that to parse data is to potentially be vulnerable to that data – especially when the parser is designed to extract context as quickly as possible. Indeed, companies such as BreakingPoint and Codenomicon have made their names building test tools to expose potential faults with DPI engines. But could anyone actually trigger these vulnerabilities? In this paper, we restart an old line of research from several years ago: The use of in-browser technologies to "tweak" Deep Packet Inspection systems.

Essentially, by controlling both endpoints surrounding a DPI system, possibly using the TCP (and sometimes UDP) socket code that plugins add to browsers, what behavior can we extract? We find three lines of attack worth noting.

First, firewalls and NATs -- the most widely deployed packet inspectors on the Internet today -- can still be made to open firewall holes to the Internet by having the browser trigger the Application Layer Gateway (ALG) for protocols like Active FTP. We extend older work by integrating mechanisms for acquiring the correct internal IP address of a client, necessary for triggering many inspection engines, we survey other protocols such as SIP and H.323 that have their own inspection engines, and we explore better strategies for triggering these vulnerabilities without socket engines from browser plugins. We also explore a *potentially* new mechanism, “Window Dribbling”, that allows an HTTP POST from a browser to be converted into a full bidirectional conversation by only allowing a remote sender to “dribble” a fixed number of bytes per segment.

Second, we (along with Robert Auger at Paypal) find that transparent HTTP proxies, such as Squid, will "override" the intended destination of browser sockets, allowing a remote attacker to send and receive data from arbitrary web sites. This allows (at minimum) extensive and expensive click fraud attacks, and may expose internal connectivity as well (HTTP or even TCP).

Third, and most interestingly, we find that active DPI's -- those that actually alter the flow of traffic between a client and a server -- all seem to expose subtly different parsers and handlers for the protocols they manipulate. These variations of behavior can be remotely fingerprinted, allowing an attacker to identify DPI platforms so as to correctly target his attacks. This capability, understood particularly in light of Felix Lindner's recent work on generic attacks against Cisco infrastructure, underscores the need for both DPI vendors to test their platforms extensively, and for IT managers to deploy critical infrastructure patches with at least as much vigor as desktop support receives today.

For remediation purposes, we recommend two lines of defense – one policy, one technical. As a matter of policy, we find the most important recommendation of this paper that **industry reconsiders patching policies as they apply to infrastructure, especially as that infrastructure starts inspecting traffic at ever higher speeds in ever deeper ways.** We are actively concerned that administrators have internalized the need to patch endpoints, but aren’t closely tracking the equipment that binds endpoints together – despite their ever increasing

intelligence. This is as much a recommendation to vendors – to build patches quickly, and to code audit and fuzz with software from companies like Breakingpoint and Codenomicon – as it is a plea to IT departments to deploy the patches that are generated. Also from a policy perspective, while this paper does recognize the need for judicious use of DPI technology, systems that are deployed across organizational boundaries have particular need for correctness. There have been incidents in the past that have led to security vulnerability across entire ISPs.

On the technical front, we defend the existence of socket functionality in the browser, recognizing that constraining all networking to that which existed in 2001 is not leading to more stable or more secure networks. We explore a solution that potentially allow firewalls to integrate socket policies into their ALG's, encouraging plugin developers to eventually join in with browser manufacturers and build a single, coherent, cross-domain communication standard. We also discuss more advanced transparent proxy caching policies, which will prevent the Same Origin Policy bypasses discussed above. Finally, we remind home router developers that browsers are *still* able to access their web interfaces from the Internet, and that this exposure can be repaired by tying default password effectiveness to either a button on the device or a power cycle.

## Tech Notes

This is a technical pre-print, intended mostly for peer review before presentation at the CanSecWest Security Conference on March 20th. It is expected, and hoped, that detailed thoughts, corrections, and (hopefully) further interesting information that should be in this paper will be sent to dan@doxpara.com before release.

The final version of this paper will include a succinct summary of each of the new technical findings in this paper here.

## Context: DPI

The fundamental design of the Internet is that of an end-to-end infrastructure. Intelligent endpoints are supposed to communicate over a "dumb" network, which is only concerned with routing traffic between these endpoints.

At least, that's the philosophy. In the real world, many types of "middleware" devices are deployed, for not unreasonable reasons. Firewalls exist to control traffic to unreliable, possibly (or even probably) insecure endpoints. Transparent proxies exist to reduce the amount of bandwidth necessary to service applications. And Quality of Service engines (or "packet shapers") are deployed to deal with particular applications that starve everything else of bandwidth. Each of these sorts of middleware need to extract context off the wire – given a set of packets, they need to determine which packets are attached to which applications, so the chosen policy can be applied. In recent years, there's been something of a sea change in terms of the new analysis engines under development. First, the level of analysis being undertaken on packet streams is getting quite a bit deeper, with complex reassembly and extraction of application context becoming standard. Second, the analysis engines have gotten much faster, inviting them to be deployed at the ingress and egress points of major networks.

Philosophies aside, this is not necessarily a good or a bad thing. Even just from a security perspective, we have problems – distributed denial of service attacks, most notably – that are difficult to impossible to solve without network-scale infrastructure.

But there are risks.

Extracting context from the wire is not necessarily an exact science. Even in the best of times, applications aren't in the business of dumping large amounts of extraneous data onto the network – they're speaking to a peer, not to an unknown network entity, so they send only the bare minimum necessary. This means that whatever policies the network seeks to apply, have to be selected according to what's already a fairly limited set of information. Nowhere is this more painfully obvious than intrusion detection, where the war between Host Intrusion Detection Systems (HIDS) and Network Intrusion Detection Systems (NIDS) has raged for years. HIDS software, living on a potentially infected system, is able to interrogate everything and anything going on inside, without worry of whatever bits are dropped on the network. But too much context can sometimes be a burden, and in case of infection, the attacker's in position to disable the protection engine before an alarm can be raised. NIDS may not have as much to work with, but it can cover more systems – especially black box appliances, into which HIDS may not be installable – and in case of a successful attack, the victim host can't necessarily disable the IDS engine.

Unfortunately, as Tim Newsham and Tom Ptacek's classic 1998 paper, "Insertion, Evasion, and Denial Of Service: Eluding Network Intrusion Detection", pointed out – there are many ways information can be sent over the wire, and it's extraordinarily difficult, if not impossible, to parse them all. Newsham and Ptacek's paper's summary concludes with the following line: "…network ID systems cannot be fully trusted until they are fundamentally redesigned."

That was 1998 – 11 years ago. To whatever degree Deep Packet Inspection technology represents a redesign, it's what we've got. It's still not enough, though – there's simply not enough context dropped on the wire to always know what's going on passively monitoring an end-to-end communication. Indeed, as my 2005 Black Ops talk demonstrated (see the Temporal IDS Evasion attack), the actual latency between a network monitor and a host is enough to cause an arbitrarily deep inspector to see something different than a host.

But as hard as it is to extract traffic when *one* endpoint is under the control of an attacker, it's even harder when *both* endpoints are conspiring against the middle. This obviously occurs if a desktop is already compromised, or if there are users outside an organizational boundary being monitored. But there's a much easier channel for dual-endpoint attack: Web browsers.

## Context: Browsers and Plugins

Software distribution is hard. It is often as much, if not more work to ship code and get it to work somewhere, than it is to write the code in the first place. This one fact has been behind the move towards shipping appliances instead of installers, and more recently, cloud-based services instead of anything at all. The difficulty of distributing software has also been behind the success of the browser. Browsers represent a tradeoff: In return for sacrificing the vast majority of the flexibility of the underlying hardware, the user gains the ability to quickly retrieve and execute arbitrary bits of software from the Internet. This is often enough to interact with centrally managed resources on the network or in a cloud. While this doesn't necessarily imply an increase in security overall – security issues in web server software are legion, and a compromise of a backend may represent a compromise of all customers rather than just one – there's no question that most software written today is written to this thin client model.

By design, the two things a browser is best at are – first, getting pixels on a screen, and second, putting bytes on the wire. While browsers are optimized for speaking HTTP, they can speak other protocols. More importantly, while browsers may *think* they are speaking HTTP, at some point they can always be made to send attacker controlled bytes. If these bytes are similar enough to the messages of other protocols, then a non-HTTP server may be tricked into accepting command from a web browser. This pattern was used to great effect in Jochen Toph's 2001 paper, "The HTML Form Protocol Attack" paper. Topf found he could manipulate a wide range of ASCII-based protocols simply through HTML forms.

Of course, the browser is capable of more than just encapsulating arbitrary protocols inside of HTTP. Over the years, several plug-ins have achieved widespread deployment, adding a cross-browser standardized Virtual Machine for speed and access to new API's that expose interesting functionality. One type of functionality has been the Browser Socket. Many web applications need to exchange a constant stream of information between a client and a server. Put simply, HTTP is *horrifyingly* inefficient at this, often add hundreds if not thousands of bytes of overhead per actual message delivered. So, ever since the beginning of the web, the Java browser plugin has offered programmers the option to get a real TCP – or even UDP – connection, accessible from within their browser deployed code.

Browser sockets were found, even very early, to have security issues. In 1996, undergraduate researchers at Princeton University noticed that the security model for Java browser sockets – intended to restrict direct connectivity to only the host that provided the Java applet to begin with – could be fairly easily bypassed, by rapidly switching the underlying IP address behind a DNS name. Dubbed "DNS Rebinding", this rapid switching would cause an applet to be loaded from one IP address, while the TCP or UDP socket went to another.

Sun fixed this issue fairly quickly, but in 2006 and 2007, a slew of new researchers – Martin Johns, Dan Boneh, David Byrne, and I all noticed that DNS Rebinding still worked, if not against proper Java applets, then against the browser itself, and specifically, the socket implementation inside of Adobe Flash. This provided a trivial way to turn the average browser into a web proxy, perhaps to internal hosts behind a firewall, but even to external servers for anonymity or false clicks on ads.

While nothing could be done about the pure-browser HTTP DNS rebinding attacks, Adobe followed the path of Sun and successfully migrated their socket code to only allow a TCP connection (Flash does not support UDP) to an IP address that had specifically opted into receiving a direct socket connection. Now, only endpoints prepared to accept arbitrary bytestreams from a browser, would be exposed. Since presumably neither internal servers nor ad servers would opt into receiving raw TCP socket communication from the outside world, both would be protected while Adobe's customers could still use sockets for their own applications. This was good.

The problem, unfortunately, is that even when Flash – or Java – or the browser itself speaks to a host, it's not necessarily or even usually the only one listening. Internal Deep Packet Inspectors, of all types, may be monitoring that communication, and applying policy in response. Those policies may not actually be appropriate, given that they're in response to content loaded in a web browser rather than full code living on the host. But the DPI, without sufficient context to differentiate the two, has to deploy the policy just the same.

But what can be done with this capability? We'll start by looking at firewall rule bypass, a field of research that was fairly active in 2001-2002 but has been abandoned since. We'll continue with manipulating transparent proxies to get around browser security policy – an attack vector co-discovered with Robert Auger of Paypal. We conclude by exploring the area we're most worried about: The reality that, merely by carefully crafting messages that may be inspected and altered by a DPI engine, an attacker can fingerprint that engine and then deliver an attack designed to exploit it.

## Attack Class 1: Firewall Bypass

### A) Application Layer Gateways: An Introduction

There is a shortage of IPv4 space, and many people have only one address but multiple Internet connected devices. To facilitate connectivity to all those devices, NAT, or Network Address Translation, is deployed on middleware devices to keep track of each client's sessions. When a response comes back from one of those sessions, the incoming packet (destined only to the NAT device) is forwarded to the internal host that spawned the request.

This process was not necessarily designed to implement a firewall. However, packets sent to a NAT are pretty much guaranteed to be dropped, unless some internal client has claimed those packets with requests of their own. Since most people want to support multiple backend devices, most home users end up with a pretty solid basic level of firewalling.

Of course, all firewalls – even NATs – need to deal with protocols more complex than HTTP. HTTP is simple to NAT: Clients make requests, servers reply to those requests on the same TCP session. At no point does the server need to know about, or deal with, the internal IP address of the client, and the server never creates its own session back to the client.

But not all protocols are as simple to NAT as HTTP. Many protocols – FTP, in particular – implement *callbacks*, in communications are initiated not merely client to server but also server to client. (Firewall engineers tend to protest that these protocols are “poorly designed”. Callbacks in fact exist in most complex software, as a basic unit of asynchronous design. Meanwhile, IP was fundamentally designed to allow end-to-end connectivity without address translation, and all over the world protocols are being shoved into HTTP frameworks for firewall compliance. Solid design principals are not doing well.)

## B) Active FTP: The Canonical ALG

Since FTP is often used to deliver important content, there is near-universal support for Active FTP in NATs. To support this, they support a form of DPI known as an Application Layer Gateway, or ALG. ALG’s watch the traffic going past, and as needed, open up additional firewall holes (sometimes called ‘conduits’) to allow whatever extra traffic needs to come in. ALG’s may also alter the command stream, inserting their own externally-facing IP address where an internal (and non-routable) IP may otherwise reside.

Of course, the only way an ALG knows it needs to open up additional firewall holes, is if it sees the appropriate commands on the wire. If a browser can successfully impersonate those commands, it can cause holes in the firewall to be pierced.

This attack class, of course, is several years old. In December of 2002, the 60th edition of Phrack (a popular zine devoted to hacking and phreaking) posted an addendum called “Java tears down the firewall”. Responding to some fairly embarrassing firewall bugs (people pushing content *into* a web server, triggering rules that allows arbitrary ports to be accessed), the Phrack authors extended the attack model to client networks as well, using Java to emulate the command channel of FTP. They observed this was a ridiculously simple attack, and indeed, looking at the protocol, it is. Simply sending the following bytes to 6.6.6.6, port 21, is enough to allow 6.6.6.6 to connect back to the web server on port 80:

```
USER anonymous
PASS foo@bar.com
TYPE I
PORT 127,0,0,1,0,80
RETR /timmeh.txt
```

…at least, on 2002 era firewalls and NATs.

This attack was independently rediscovered by Florian Weimer in July of 2005, in “The Java/Firewall Vulnerability”. We then rediscovered it ourselves. More than any other bug I know of, people keep bumping into this, and promptly forgetting it.

Interestingly, there is a flaw. Under testing, a surprising number of devices actually do care about the source IP declared within the FTP command channel. If the IP is incorrect, the ALG doesn’t fire. The original code from Phrack #60 does try to take this account, with this line:

```
String me = InetAddress.getLocalHost().getHostAddress();
```

However, this now returns 127.0.0.1, which will not elicit ALG behavior.

C) Recovering Local IP Addresses

We explored whether it was possible to still recover the local IP of a browser, and found three ways:

First, we could simply brute force. Most home users are behind firewalls within RFC 1918 space, and specifically, inside the ranges:

192.168.0.1-254
192.168.1.1-254
10.0.0.1-254
10.0.1.1-254

This however does not work universally; there are large RFC 1918 networks and there are even desktops on globally routable IP space. Probably the best mechanism known for determining internal IP address comes from Lorenzo Baloci, maintainer of the aptly named http://amibehindnat.com, who found that if, instead of the above code, he did something akin to the following, he'd get the local IP of the caller:

```
Socket s = new Socket(www.badguy.com,80);
String me = s.getLocalAddress();
```

This will indeed return the local address of whatever interface is used to connect to the attacker. This is dependent on Java being installed, however – meanwhile, FTP is just as vulnerable to Flash emulating its control channel, assuming Flash can discover the local address. But Flash has no obvious API's that will leak this (though there may be something floating around RTFMP).

Flash does, however, have access to the resources of the underlying browser, and the underlying browser stack does support FTP on its own. So, theoretically, one could ask the browser to retrieve ftp://ftp.badguy.com/timmeh.txt, and the bytes sent would be something akin to:

```
USER anonymous
PASS foo@bar.com
TYPE I
PORT 192,168,0,194,10,192
RETR /timmeh.txt
```

…reflecting an internal IP address of 192.168.0.194. There are two problems with using Active FTP in this way. First, the browser may no longer support it: Modern versions of Firefox have gone Passive FTP only. That does not make Firefox immune, at least by proxy: Firefox is able to play media formats that, on Windows, will launch Windows Media Player. Windows Media Player will attempt to load content via FTP, using the same Active FTP supporting stack as Internet Explorer.

However, independent of FTP stack, there's a larger, more obvious problem: You're trying to discover the local IP address, so you can use it to trick the ALG into opening up conduits. But when you emit the local IP address, it is within an Active FTP session, so the ALG dutifully wipes out the browser supplied local IP and replaces it with its own. Is there a way around this?

Potentially, there are two. The first is simply to take advantage of the fact that most of the FTP ALG's out there are *port-specific*, meaning they'll only function if the TCP port carrying the FTP connection is equal to 21. FTP URLs can of course apply to any port, so ftp://ftp.badguy.com:20021/foobar.txt will cause the browser to initiate a FTP command channel on 20021/tcp. The ALG won't trigger, and thus the actual internal IP sent in FTP's PORT command will leak through.

The second path is more interesting.

D) Window Dribbling And Other Stream v. Segment attacks

TCP is a stream oriented protocol, meaning it transmits a stream of bytes that may be sliced and diced across any number of packets. Any boundaries between one command or another must be in-band; the number of packets used to transmit the message is supposed to have no bearing on the actual message parsed. However, it's somewhat resource intensive to fully reassemble IP packets and TCP segments into a full stream that can be understood and analyzed. And so many firewalls don't, assuming that since applications tend to dump their FTP commands into a TCP buffer all at once, that that's what's going to show up on the wire. This led to all sorts of interesting attacks between 2000 and 2002, playing with the boundary between TCP's streams and IP's packets.

One particular post that comes to mind is Mikael Olsson's from October 2002: "Firewalls that support FTP without fully reassembling the FTP command channel can have their rulesets bypassed. Again." His strategy was based around Partial Acknowledgements. Suppose a byte stream such as:

ABCDEFGHIJKLMNOP

Mikael's concept was that one could acknowledge receipt of A through K, leaving L through P to be retransmitted by the other node. If L through P was, for example, PORT 1,2,3,4,0,80 , this might cause a firewall to open up a hole on port 80.

We believe there is an interesting and unique variant of this attack. Instead of using Partial Acknowledgements, we employ something that can be called "Window Dribbling". A TCP endpoint is able to declare exactly how many bytes it's able to receive from another endpoint. The window size is a 16 bit integer, and is usually set to reasonably large size to allow many individual packets to be in flight. However, the window can also be set very small, to only allow a few bytes to come in.

What this means is, for an arbitrary message sent by TCP endpoint, the *receiver* of that message can directly control its packetization. This means there's a fairly straightforward way to evade an ALG's FTP filter: Still allow the Active FTP transaction to function on port 21, but split the PORT command across multiple packets. Imagine a FTP sequence looking like:

```
Command 1: USER anonymous
Command 2: PASS foo@bar.com
Command 3: TYPE I
Command 4: PORT 192,168,0,194,10,192
Command 5: RETR /timmeh.txt
```

Since these are all pretty small messages, generally they'll each consume one TCP Segment… unless the segment containing "TYPE I" is acknowledged, but with a window size of 3. That will cause Command 4 to be split – with the first segment containing:

POR

…then, an acknowledgment w/ a window of 6 bytes will generate:

T 192,

…at which point, the rest can be acknowledged, and as long as the firewall can't reassemble three segments, this will not hit the ALG and the FTP connection will pass through, leaking the internal IP.

There have been games with trying to limit message size before – see work done on TCP Maximum Segment Sizes – but it's interesting that we can use Window size, specifically, to *prevent* parsing of ALG data.

Interestingly, as DPI technologies have scaled up, and gotten much faster, stacks in firewalls have gotten more and more capable of assembling data across any number of segments. This opens up an interesting problem – sometimes, there are two firewalls: A network firewall, which is highly intelligent and handles ALG across segment boundaries, and a host firewall, that might not. For example, some host firewalls will only interfere with PORT commands that are emitting commands in a single segment. Either by using Window Dribbling on the receiver, or by scripting buffer flushes into our sender, it's straightforward to bypass these host firewalls, while still opening up a hole in the network firewall that can assemble across segments.

Window Dribbling attacks get particularly interesting due to their ability to allow browsers – independent of sockets – to effectively emulate complex negotiated protocols. Consider again Topf's HTML Form Attacks. These attacks are based on dropping a load of data, all at once, at a particular server. But most protocols are conversations, not blobs – and many inspectors are expecting to see those conversations. By dribbling things out, we can emulate a complex bidirectional communication from a client that's really just emitting an HTTP POST as per Topf.

## E) Previous Responses To HTTP PORT-based attacks

Of course, Topf's research came out in 2001, and quickly spawned a set of defenses. Both Firefox and Internet Explorer block a wide range of ports, specifically to prevent HTTP POSTs (which, unlike socket calls, can by design be delivered to any IP address) from being delivered to ports that might not actually have HTTP listeners on them. According to Mozilla, these ports include:

| Port | Service | Port | Service | Port | Service |
|---|---|---|---|---|---|
| 1 | tcpmux | 102 | iso-tsap | 530 | courier |
| 7 | echo | 103 | gppitnp | 531 | chat |
| 9 | discard | 104 | acr-nema | 532 | netnews |
| 11 | systat | 109 | POP2 | 540 | uucp |
| 13 | daytime | 110 | POP3 | 556 | remotefs |
| 15 | netstat | 111 | sunrpc | 563 | NNTP+SSL |
| 17 | qotd | 113 | auth | 587 | submission |
| 19 | chargen | 115 | sftp | 601 | syslog |
| 20 | ftp data | 117 | uucp-path | 636 | LDAP+SSL |
| 21 | ftp control | 119 | NNTP | 993 | IMAP+SSL |
| 22 | ssh | 123 | NTP | 995 | POP3+SSL |
| 23 | telnet | 135 | loc-srv / epmap | 2049 | nfs |
| 25 | smtp | 139 | netbios | 4045 | lockd |
| 37 | time | 143 | IMAP2 | 6000 | X11 |
| 42 | name | 179 | BGP | | |
| 43 | nicname | 389 | LDAP | | |
| 53 | domain | 465 | SMTP+SSL | | |
| 77 | priv-rjs | 512 | print / exec | | |
| 79 | finger | 513 | login | | |
| 87 | ttylink | 514 | shell | | |

|  |  |  |  |
|---|---|---|---|
| 95 | supdup | 515 | printer |
| 101 | hostriame | 526 | tempo |

Internet Explorer has a similar list of ports, though perhaps smaller.

It should be obvious even to a casual observer that attempting to predict, based on just a port number, what services really don't want to be hearing from a web browser is not a defensive strategy that can scale. There's always another port that might be missed, and frankly, there's always a possibility that there really is a web server on 6000/tcp or whatever else. These sorts of port blocking strategies are reasonable attempts at providing best effort security, but it's ultimately an unstable approach to the problem.

F) Other Protocols Besides FTP

Where things get interesting, both when Window Dribbling an HTTP POST, and with sockets themselves, is that there are indeed many protocols out there, and a browser literally can't know or predict all of them. Some of these protocols actually behave similar to FTP, in that traffic on a command channel will get picked up, analyzed by an ALG, and dropped on the wire. For example, DCC, the direct host-to-host communication protocol generally used by Internet Relay Chat (IRC) for file transfer, was exploited quite similarly to FTP in 2001 by Michal Zalewski. IRC runs over 6667 – a port missing from the above list, coincidentally. But there are other protocols out there. In terms of what's seen out there, in terms of ALG's inside of firewalls:

| **Deployment Status** | **Protocol** | **Starter Port** | **Conduit Port** | **IP In Conduit** | **Been Publicly Poked At** |
|---|---|---|---|---|---|
| Universal | Active FTP | 21/tcp | */tcp | Yes | Yes |
| Common | H.323 | 1720/tcp | */tcp, */udp | Yes | No |
| Common | SIP | 5060/udp | */udp | Yes | No |
| Common | PPTP | 1723/tcp | GRE only | Unknown | No |
| Corporate | RSH | 514/tcp | */tcp | No | No |
| Corporate | MSRPC | 135/tcp | */tcp | Unknown | No |
| Corporate | SQL*Net | 1521/tcp | */tcp | Unknown | No |
| Linux | SANE | 6566/tcp | */tcp | Unknown | No |
| Linux | TFTP | 69/udp | Fixed port | Unknown | No |
| Linux | IRC DCC | 6667/tcp | */tcp | Yes | Yes |
| Linux | Amanda | 10080/udp | */tcp | Unknown | No |

(A future version of this paper will include details on a few other protocols for which handlers have been seen, but which we don't yet have good specs on: Skinny, RTSP/MMS, and ILS for example.)

We do seem to have four classes of deployed protocols: The universal class – FTP – is on pretty much everything, home and corporate. The common class – H.323, SIP, and PPTP – tend to show up on most home and corporate gear. Corporate class ALG's include RSH, MSRPC, and SQL*Net. And then there are handlers that come with Linux, which indeed do show up in some environments, but are the rarest.

Each protocol has a starter port – and more importantly, a choice whether it runs over TCP or UDP. Only Java can be made to create sockets to UDP starters. Flash, and potentially the browser HTTP engine itself, are limited to ALG's that listen on TCP ports.

It is worth mentioning that Silverlight, while it does support TCP sockets, only allows connections to 4502/tcp through 4534/tcp. To be fair, firewall vendors have stated privately that in the long view, they see their port-specific filters being dropped in favor of DPI engines that find control channels on any port. But whatever long-term changes DPI brings, none of the above ports fall into the 4502/tcp to 4534/tcp range.

A number of protocols, like FTP and DCC, explicitly declare the IP address they'd like communications to return to. This can potentially allow an attacker to declare IP addresses *different* than the local IP during his spoofed FTP or DCC connection, granting him the sort of generic behind-the-firewall access that was possible back when browser sockets could be DNS rebound to any IP address. Both FTP and DCC were publicly thrashed years ago for exactly that sort of vulnerability, however, so it's (hopefully) doubtful there aren't too many firewalls left with fully vulnerable ALG's of this nature.

Surprisingly, RSH – the insecure predecessor of SSH – is surprisingly supported amongst corporate-class firewalls, and when emulated, will grant access to arbitrary backend ports. RSH is old enough that there is no established RFC for it. However, its protocol is almost certainly nearly identical to that for its cousin, rexec:

The rexec protocol works as following :

1. Client makes TCP connection to REXEC port (512).
2. Client sends TCP port number (decimal ascii, null-terminated) of stderr port. If the first byte is a NULL, then server won't make any stderr connection - skip step 3.
3. Server makes TCP connection to client's stderr port
4. Client sends target username (null-terminated).
5. Client sends target password, NULL, remote command, NULL, and then command's stdin, followed by a FIN.
6. Server sends one null byte (=no error), or a non-null byte, followed by error message(s).
7. Server sends output of command.
8. Server sends FINs on stdout, stderr connections.

There is no IP address declared for RSH *or* REXEC's stderr channel, and so there's no opportunity for arbitrary behind-the-firewall access. However, the VoIP protocols – SIP, and particularly H.323 – *do* expose non-IP specific ALG's, possibly by design. H.323 is a complicated protocol to NAT, with many callbacks that need conduits in order to function. Looking at the Linux implementation, there are fully *nine* conduit types that can be opened:

1. An RTP stream, to any UDP port on the caller
2. An RTCP stream, to any UDP port on the caller
3. A T.120 stream, to any TCP port on the caller
4. A H.245 stream, to any TCP port on the caller
5. A Q.931 stream, to any TCP port on the caller
6. A "GCF" stream, to a potentially filtered UDP port
7. A "ACF" stream, to a potentially filtered address
8. A "LCF" stream, to any TCP port on **any** address
9. A H.225 *call forwarding* stream, to any TCP port on **any** address

The code for this in the H.225 code is here:

```
/* Read alternativeAddress */
if (!get_h225_addr(ct, *data, taddr, &addr, &port) || port == 0)
        return 0;
```

```
/* If the calling party is on the same side of the forward-to party,
 * we don't need to track the second call */
if (callforward_filter &&
    callforward_do_filter(&addr, &ct->tuplehash[!dir].tuple.src.u3,
                          nf_ct_l3num(ct))) {
        pr_debug("nf_ct_q931: Call Forwarding not tracked\n");
        return 0;
}

/* Create expect for the second call leg */
if ((exp = nf_ct_expect_alloc(ct)) == NULL)
        return -1;
nf_ct_expect_init(exp, NF_CT_EXPECT_CLASS_DEFAULT, nf_ct_l3num(ct),
                  &ct->tuplehash[!dir].tuple.src.u3, &addr,
                  IPPROTO_TCP, NULL, &port);
exp->helper = nf_conntrack_helper_q931;
```

Similar code shows up for LCF handling:

```
if (!get_h225_addr(ct, *data, &lcf->callSignalAddress,
                   &addr, &port))
        return 0;

/* Need new expect for call signal */
if ((exp = nf_ct_expect_alloc(ct)) == NULL)
        return -1;
nf_ct_expect_init(exp, NF_CT_EXPECT_CLASS_DEFAULT, nf_ct_l3num(ct),
                  &ct->tuplehash[!dir].tuple.src.u3, &addr,
                  IPPROTO_TCP, NULL, &port);
exp->flags = NF_CT_EXPECT_PERMANENT;
```

**Having spoken to one Linux developer, it is clear this is not buggy behavior.** Rather, this is quite intentional, specifically required for the ALG to function correctly. It is indeed difficult to imagine how call forwarding could work, without actually proxying traffic to a different backend phone than the one who actually made the call.

G) Per-Firewall Caveats

Of course, it's possible not all firewalls support call forwarding, or in the case of at least one firewall vendor, enable H.323 handling for all backend devices. Unlike FTP, H.323 support is going to vary from environment to environment – even if there is a module, it may very well not be enabled at all.

Also seen in some devices is an unwillingness to allow NATting back to ports below 1024. This harkens back to the era when all "services" lived on the Unix-root controlled low ports. There is actually some utility here: Almost all operating systems will only reserve sockets to userspace from 1025 and higher, and many services have simply migrated to 80/tcp because – ironically – it was the easiest to get through firewalls. However, valuable services will always live on ports higher than 1024, such as Terminal Services, X11, and NFS. This is a useful but not comprehensive mitigation.

H) Conclusions

The point of this section is not to point out that firewalls have bugs. The point is to show that the most widely deployed Deep Packet Inspection engines – the ALG's inside of firewalls – do indeed have behavioral traits that are not necessarily expected.

But firewalls are not the only DPI engines out there. There's another type of engine, that though not as commonly deployed, does find itself in front of very large networks: Transparent Proxies.

## Attack Class 2: Same Origin Bypass Via Transparent Proxies

### A) Shared Work

At this point, it may be good to point out that this section of this paper was independently discovered by Robert Auger of Paypal, Inc., who has his own paper on the subject entitled "Socket Capable Browser Plug-ins Result In Transparent Proxy Abuse". Robert and I have been working together on these findings, and his paper should be seen as the canonical document on socket/proxy interactions. However, our papers have a somewhat different focus, so I'll still establish some context in this document.

### B) Same Origin Policy: An Introduction

Though not at all the Internet's only application, the web is for all intents and purposes *the* face of the Internet for a large percentage of its userbase. Web browsers, as described earlier, sacrifice most of the capability of underlying hardware in return for the ability to (relatively) safely retrieve code from the Internet. The code that is run actually has a very interesting security policy associated with it. On the web, the DNS name of the web site that provides script actually represents the *principal*, or identity, under which that script is executed. For example, when code loaded from http://www.google.com opens a window, only other code loaded from http://www.google.com can access the resources of that window. Code from http://www.microsoft.com cannot.

The Same Origin Policy's allowances are as important as its restrictions. Microsoft is allowed to create a link to Yahoo, and even to HTTP POST to a Yahoo server. (For such a cross domain post, it's not allowed to read back from code the result of that POST, though the user is navigated to the destination page.) Far from being simple, Same Origin Policy is actually a highly nuanced, remarkably effective security model that allows multiple providers of untrusted code to share the same basic user experience, without exposing the contents of one site to the code of another.

By contrast, on effectively all user platforms, any software running as a user can manipulate any other software running as that user. Process-level, or intra-user security, has been difficult to deliver. A notable system that tried – systrace, which filtered system calls from user to kernel – was defeated so thoroughly, it was effectively permanently removed from operating systems that had deployed it.

One fair critique of the Same Origin Policy is that it was never fully designed in advance; rather, it is the result of a decade of hard choices by the relatively small set of browser manufacturers. This lack of design has led to a lack of documentation. Seeking to address this, Michal Zalewski recently published his "Browser Security Handbook", a detailed guide that tries to synchronize expectations about the browser across users, web developers, browser manufacturers, and importantly, plugin authors.

Plugin authors, such as Adobe for Flash and Sun for Java, have had a rough couple of years keeping up with expectations regarding their behavior vs. behavior expected within the browser. For example, Amit Klein found in July 2006's "Forging HTTP request headers in Flash" that Flash had slightly different header setting policies than the browsers it was hosted inside. This allowed an attacker to bypass assumptions that web developers had made regarding the capabilities, or lack thereof, of a browser. More seriously around that time, Amit, along with Chaim Linhart, Ronen Haled, Steve Orrin, and Watchfire Security released a paper by the name of "HTTP Request Smuggling", in which browser/plugin behavior was analyzed in the context where there wasn't in fact an end to end connection between client and server. Instead, a *proxy* – a device sitting between the web client and the HTTP server – receives requests explicitly from a client, and this new device executes HTTP requests on its behalf.

What Amit et al found was that there were some extraordinary vagaries to the HTTP protocol that caused some pretty bad things to happen inside of proxy caches. The Request Smuggling paper was written without knowledge of 2007's discovery that browser sockets could be used to trigger all of its attacks remotely, and that bug was fixed fast enough that it never really needed to. Browser sockets relatively quickly became only able to communicate with an erstwhile attacker.

But browser sockets are still around, and while they won't connect to just any address, they may not need to. A certain type of proxy – a *transparent proxy* – doesn't care where a client is trying to connect to. A browser, via Java or Flash, can create a perfectly valid socket connection to 6.6.6.6 on 80/tcp, and that TCP socket will not actually go where it was asked. Instead, it will enter the proxy, which will complete the HTTP request according to whatever policy it has.

This poses something of a problem. A web page is actually allowed to include by reference content from anywhere on the net. However, according to Same Origin Policy, that which is shown to the user isn't necessarily allowed to be read from script. So, for example, one can embed an image from a foreign URL, but the bits of that image cannot be read back via script. Such nuanced policy does not exist when the client is not actually the browser itself, but a socket injected via a plugin. The socket got bytes, the plugin gets bytes, the attacker can read the bytes.

Unfortunately, from the transparent proxy's perspective, there's no way to tell this has happened. The bytes on the wire are exactly the same.

C) Basic Proxying

Consider a minimal HTTP 1.1 request for http://www.google.com. This will normally take the following form:

```
GET / HTTP/1.1
Host: www.google.com
```

However, it turns out the following alternate syntax is also acceptable, back from HTTP 1.0:

```
GET http://www.google.com HTTP/1.0
```

Normally, a client will initiate a TCP connection to www.google.com's IP, on port 80, issue one of the two above requests, receive Google's content, and render it under the www.google.com Origin. When a transparent proxy is in the way, the client believes it's doing exactly this. However, the network infrastructure in a transparent proxy case has been modified, perhaps by inspecting the TCP destination port and seeing it as 80, the port for HTTP. In this case, the original destination IP is ignored as the packet is tunneled over to the transparent proxy (generally over GRE, WCCP, or via MAC Rewriting).

At this point, the proxy sees a request for Google. It has two choices:

1) It could recover the original IP address that the client was speaking to, and send a HTTP request to that IP address. It could then return Google's content.
2) It could ignore the original IP address, do a DNS lookup for www.google.com against its own name server, and return the Google content from the IP address returned

At this point, it's important to realize why the proxy is there in the first place. While proxies are often in place to implement content filtering, their primary role is often to make things faster, and to reduce the amount of bandwidth used by the organization, by allowing HTTP caching across the entire organization. If the request is sent to the client supplied IP address, then the content returned can only be cached against the combination of that particular IP and the Host in question. Otherwise, it would be trivial to poison the cache's version of Google – simply ask 6.6.6.6 for www.google.com, have your traffic to 6.6.6.6 get hijacked by the proxy, and watch not only the cached version of 6.6.6.6's www.google.com contain your false response, but *all* IP's idea of www.google.com as well.

The attack above is obvious enough that we have not yet found any transparent proxy servers vulnerable to such a trivial HTTP Cache Poisoning attack.

What we *have* seen, however, is that most (not all) HTTP proxies follow the second caching policy above. Whatever IP address the client was seeking is ignored; instead, the proxy does its own DNS lookup, retrieves IP addresses that are at least more trustworthy than what some random client might gin up, retrieves Host content from those IP's, and caches the results as per that Host.

This would seem to work well, except it doesn't at all handle the case where there actually *is* a request sent to 6.6.6.6 for http://www.google.com, because a browser socket synthesized the exact request for Google and sent it to an attacker's IP. In that case, the attacker gets to send and receive arbitrary requests to Google from the victim's IP.

D) Impact

At first glance, the impact of being able to access Google, or Microsoft, or any location proxied off some random browser and its transparent proxy is limited. Indeed, since the connection goes through a socket, and not through the browser stack itself, traditional threats like Cross-Site Request Forgery (CSRF) aren't in play because the browser socket won't have access to login credentials or cookies for various sites. No credentials, no ability to impersonate the user on foreign sites, no damage.

This entirely misses the problem of Click Fraud.

Online advertising is (at least according to 2008 numbers) a multi-billion dollar business. One of the more common ways online ads are paid for is Pay Per Click: As long as the right packets hit the ad server, a small amount of revenue is realized.

This hack – like the DNS rebinding attacks against browser sockets in 2007 – allows the right packets to hit the ad server. The revenue realized goes to wherever the attacker wants.

Google claimed in 2007 they had a $1B Click Fraud problem. In 2008, Jeremiah Grossman and Adrian Evans, in their talk, "Get Rich Or Die Trying", put some evidence to the claim by displaying a check to a click fraud scammer in excess of $967,000. (This was apparently the second check sent by Google. The first, for over $1M, was rejected by the bank due to paperwork involved in cashing it. The scammer was asked to request a smaller check from Google. Google paid.)

Besides enabling Click Fraud, it may be possible for an attacker to access internal resources behind the firewall. If the transparent proxy is set up behind the firewall, then simply issuing commands akin to:

```
GET / HTTP/1.1
Host: internal
```

or:

```
GET http://internal HTTP/1.0
```

…will result in internal access.  In fact, Robert Auger has found at least one device that allows the following HTTP syntax to be used:

```
CONNECT www.google.com:80 HTTP/1.0
```

This HTTP syntax, mainly intended for allowing a client to get a raw TCP socket so SSL can be completed via an explicitly configured proxy, apparently occasionally also works for transparent proxies as well.  Networks with this configuration unfortunately end up with the entire browser socket lockdown obviated – the plugin does all the work to make sure it's speaking to an IP that opted into receiving traffic, but the network hijacks that traffic and redirects it based on data in the payload.

Luckily only one device was found where the above worked.  It was however fairly common to find devices where CONNECT would function when the destination port was 443, the port for HTTPS.

Along similar lines, proxies that don't support CONNECT syntax in transparent mode, may yet support the following syntax:

```
GET http://www.google.com:21 HTTP/1.0
```

In this case, an attacker might be able to drop HTTP "POSTs" on ports that the browser itself natively refuses to POST to.  This could allow all of the attacks in Toph's work, as well as the mechanisms described in Attack Class 1.  No bug is so good that another bug cannot make it better.

## E) Other Protocols

Relatively few other protocols are even proxyable, let alone transparently proxied to any significant degree.  We are aware of some SMB WAN accelerators out there; it is possible that some of them may be exposed in interesting ways to malicious endpoints conspiring against the middle.  However, our communication with a number of major vendors with SMB WAN accelerators yielded no obvious vulnerabilities.

Interestingly, the most common aggregation technique in the field is not HTTP or SMB but SMTP. To prevent SPAM, it's very common that ISP's hijack traffic sent to 25/tcp.  Since this is specifically designed as a SPAM fighting mechanism, I don't necessarily see an attack possible, though this does bypass the browser plugin's desire to only send traffic to nodes that explicitly accept the traffic.

## F) A Unifying Theme

Ultimately, in both connectivity-expanding scenarios, we have Deep Packet Inspection responding to traffic on the wire in a way that doesn't actually reflect what's going on with the backend host.  In the ALG case, we see a firewall assuming a browser is in fact a FTP, or VoIP, or RSH client.  In the HTTP Proxy case, we see a proxy assuming a browser socket is the browser's own HTTP stack.  There is simply insufficient context on the wire to know exactly what's going on in the client, but for scalability reasons, what is on the wire is all that is available to analyze.

Another key theme between the two attacks is, to a very real degree, their effectiveness varies from device to device.  This is because everybody's implementation is unique, and different.  This opens up a whole new realm of attack:  Direct exploitation of middleware, after fingerprinting the make and model via differences in the Active DPI inspectors.

## Attack Class 3: Fingerprinting Active DPI

### A) Context

There is an old quote in network engineering: “Be conservative in what you send, and liberal in what you accept.” The general design principle is to be as simple as you can be for anyone else to understand what you say, but to accept as weird stuff as you possibly can when accepting messages because who knows how bad other people’s code is. At some level, this philosophy ends up *discouraging* reasonably standardized implementations, because nobody is actually penalized for making a broken implementation as their code still works (everyone else is simply special casing their particular weirdness). But still, this approach tends to be the general model taken within network code.

However, what is the precise definition of conservative and liberal (besides a phrase that’s vaguely surprising to see in a technical brief)? Put another way, when dropping bytes on a wire, how simple is enough? And when parsing packets, how weird is too weird? The answer is, there is no answer. Every implementation is a little different. These differences can be detected, and thus used to remotely determine what implementation is in play.

At this point, it’s probably important to defend the authors of fingerprintable code. There’s really not much they can do. Fingerprinting is often considered the canonical Won’t Fix design bug ,for a reason: No matter how pedantic a standard, there is *always* room for implementation variance. The standard for shipping software is always “does it work well enough”. If we’re lucky, that means “is it secure”. Asking for an implementation to not only work, to not only be secure, but to also have no remotely visible differences from the competition is an impossible request, especially since some amazingly subtle differences – such as how long it takes a server to reply to a given request – or even market-driven differences – how many requests per second a service can handle – are by their nature unique fingerprints too.

And so there’s a long history of network fingerprinters. Ofir Arkin, back in 2001, demonstrated that merely analyzing how a given IP address responded to ICMP packets was enough to get a pretty good idea what operating system you were speaking to. This strategy was eventually extending by the author of NMAP, Fyodor, into the realm of analyzing not only ICMP behavior but TCP as well. Indeed, it turns out that fingerprinting of endpoint operating systems is possible even without special probes, but just with entirely normal traffic. This fact was first noticed by the subterrain crew’s siphon, and eventually perfected by Michal Zalewski with p0f.

The history of being able to fingerprint middleware is much hazier. Sometimes, firewalls expose endpoints to the outside world – IKEv2 for IPSec VPN, for example. To fingerprint the device in the middle, one merely connects directly to it and sees who responds. My 2008 RSA talk explored a variant of this mechanism, when I used DNS Rebinding against a browser to remotely connect to a home user’s router, which often a) has a predictable password, b) is at a predicable IPv4 address, and c) by necessity is exposed to the browser regardless of whatever wireless cryptography is in play.

But most middleware, especially the sort of Deep Packet Inspecting middleware being developed now, exposes no direct endpoints to be queried. Determining the presence or the absence of the software, or its precise type, would need to be inferred only from the effects on streams flowing through it. Relatively little (public) work has been done analyzing this problem. I had a bit of this in my 2002 Black Ops talk, where I noted that a host behind a Cisco PIX firewall would change the IP TTL when rejecting a connection, while if the PIX itself implemented the rejection, the IP TTL would stay the same. (PIX’s are always pretty easy to recognize; SMTP through a PIX is highly and visibly sanitized.) I also talked a little about detecting Intrusion Protection Systems, based on their willingness to emit Reverse DNS Requests for attacker controlled addresses in my 2005 Black Ops talk.

But there is much more out there for fingerprinting endpoints than there is for fingerprinting middleware. Fyodor's firewalk, released in XXX, represents a pretty reasonable strategy for locating the precise hop at which a firewall (though not *which* firewall) is filtering IP traffic. Probably the most successful middleware fingerprints happen at the application layer, where HTTP Load Balancers simply announce themselves in extra headers, and as a researcher by the name of SoWhat found anti-virus mail gateways commonly announce their precise make and model information upon rejecting a harmless ancient virus.

This still leaves large amounts of DPI invisible, yet active – especially when the DPI lives on the client, and we are not able to directly connect to systems behind it. The key to detecting this middleware appears to be to use the browser. Using the mechanisms discussed earlier in this paper, it's possible to emulate various protocols speaking through DPI systems. Even better, it lets you precisely modulate how closely to a protocol spec you are, or not adhering. As long as the DPI in the middle is *active* – meaning it causes some sort of change, visible to either endpoint, based on whether it was or was not able to parse a message – it may be possible to fingerprint the DPI, possibly down to the version.

B) HTTP

The easiest form of DPI fingerprinting is simply an analogue of existing strategies for detecting Load Balancers. When traffic transits through a HTTP proxy like Squid or Bluecoat, extra headers tend to be attached – a Via header for Squid (complete with version information), a X-Bluecoat-Via header for Bluecoat, and so on.

By simply generating a static request, and comparing what gets sent, to what gets received, it's fairly straightforward to identify the existence of most proxies. Even better, since browsers actually emit almost entirely predictable HTTP requests, basic proxy detection can be done entirely passively, simply by analyzing headers in everyday HTTP traffic.

We can, however, learn much more about proxies in play by generating our HTTP queries directly from a browser socket. As discussed earlier, while browser sockets will now only go back to the attacker's IP, transparent (though not explicit) proxies will often hijack that traffic anyway. At this point, we can issue a wide variety of somewhat pathological HTTP queries, and based on which queries yield which results, we can trivially determine what proxy we're going through. Variables that can be changed include:

1) If there are *multiple* Host headers in a HTTP request, which one will win?
2) What if there's both the GET http://www.foo.com syntax, *and* a Host header?
3) Are headers sorted, or re-ordered before delivery to a server?
4) If there are duplicate headers, are any suppressed or merged together?
5) Is any parsing case insensitive?
6) Is any parsing whitespace sensitive?
7) What happens to cache policy, when contradictory or inappropriate fields are inserted in the request, or in the response?

Cache policy is of particular importance, because a *second* (browser-socket based) request for the same URL will return very specific content based on the peculiar intersection of how the browser appeared to ask for something, how the server appeared to respond, and how "happy" the client was with both. Both *whether* anything is returned from cache, and *exactly what* is returned, will necessarily vary from

Indeed, through the lens of this paper, Klein et al's HTTP Request Smuggling paper back from 2005 becomes relevant once again – not merely because it turns out all of their attacks were fully realizable when written given nothing but an attacker with access to an ad network – but because, even today, it points the way towards many fingerprinting strategies. For example:

1) Given a request with multiple Content-Length headers, SunOne Proxy 3.6 will cache based on the second one.
2) Checkpoint FW-1 has a well known set of URL filters, which it will refuse to allow to reach the outside world. For example, it will block GET /page.asp?cmd.exe HTTP/1.0. By intentionally GETting this, FW-1 can be detected (or at least ruled out).
3) Delegate 8.9.2 will respect a Content-Length in a GET *request* (Content-Length usually only shows up in a POST request or an HTTP response), altering cache semantics.

The paper has further obscura on Microsoft ISA/2000, Squid, Oracle WebCache, and pretty much every other proxy server. While it's concerned with the much more severe problem of HTTP cache pollution – problems which presumably have been fixed since 2005 – the paper is something of a roadmap for how to build a comprehensive HTTP fingerprinter.

C) Enter The Firewall: Looking at ALG's

Transparent proxies have a definite presence, but for widespread deployment, nothing is more common than Application Layer Gateways inside of NATs and Firewalls. On the surface, identifying any ALG is as simple as just doing what we did to HTTP with other protocols. However, there are some interesting quirks:

1) While a transparent proxy is effectively a complete HTTP stack, with complete and total awareness of HTTP semantics, ALG's tend to comprehend as little of the protocol in question's semantics as possible – just enough to figure out what conduits need to be opened up, and no more. That means there's no cache to interrogate, and no complex reordering likely even on protocols that look like HTTP, such as SIP.
2) Since ALG's tend to be very simple, fragmentation – either at the segment level, with TCP, or at the IP level, with UDP – may or may not suppress triggering the ALG.
3) ALG's do two things once they've successfully parsed a protocol: First, they open a listening port on the external interface, the existence of which can be tested. Second, they "mangle" the protocol to express a new IP address, the precise nature of which can be monitored.
4) ALG's can and will detect invalid protocol messages. What happens when they do detect an error differs from implementation to implementation, however. Some responses seen in the field:
    a. Session Close – the TCP session is shut down entirely when an invalid message is seen. (Depends on a TCP session.)
    b. Session Lock – the TCP session becomes unresponsive when an invalid message is seen (Also depends on a TCP session.)
    c. Message Drop – The particular message containing the verboten material is just not sent, but more messages can be sent.
    d. Message Pass – The particular message doesn't trigger the ALG, but otherwise passes through it just fine
5) Transparent proxies do have situations in which they're chained – see Robert Augur's paper for details. ALG's are often chained for FTP though, where both host and network firewalls both need to allow transit.

Given these quirks, the attack strategies are similar – add and remove whitespace, have valid or incomplete IP addresses, duplicate or suppress necessary fields, and so on.

Policy Remediations

A) Patch Infrastructure.

Without question, **the most important recommendation this paper is for the industry to reconsider patching policies as they apply to infrastructure, especially as that infrastructure starts inspecting traffic at ever higher speeds in ever deeper ways.** The simple reality is that IT policies tend to be very careful about desktop and server patch levels, and nowhere near as careful about the infrastructure that glues desktops and servers together. Some of this comes from historical experience – when the worms really hit in 2003, it wasn't the firewalls that crashed, nor was it the routers, but it was desktops and servers that were hit.

There is also some degree of fear around patching network infrastructure: If a patch fails on a desktop, that desktop is out of commission. If a patch fails on a firewall, *every* desktop loses connectivity to the Internet. And patch failure is not unimaginable: Automatic updates are rarely if ever found in the infrastructure world, and in the case of large customers, firmware is so heavily customized that the sort of extensive testing and rapid deployment of patches we enjoy in the desktop realm is fairly unimaginable.

So infrastructure is *just not being maintained all that well*. And, independent of this paper, we are starting to see a theme towards exploiting that fact.

First, there was my 2008 DNS vulnerability. A fairly straightforward bug in its own right, what actually mattered about it was how poorly the "end-to-end" application ecosystem was actually able to deal with a compromised infrastructure. The simple truth is far too much depends on DNS, and indeed the entire underlying network, being secure – authentication is not ensured, it is simply assumed. We do have some reason to be hopeful: IT departments around the world rallied strongly to patch their DNS infrastructure, despite struggles to first *identify* what needed to be patched, and then again to make the patches work despite interference from another sort of infrastructure, the firewall. And DNSSEC in fact looks like it can, in the medium term, address some of our fundamental failings to authenticate.

But my attack was but one of four that came out in 2008 that allowed an attacker to hijack the underlying network. From SNMPv3, to BGP, to WPA2 (XXX Add Names), many researchers found core flaws in infrastructure that endpoints are just not prepared yet to absorb.

The second finding that really underscores the need to update patching policy is the December 2008 announcement of generic attacks against Cisco infrastructure. For a long time, the daunting number of customized images of Cisco firmware was considered a deterrent to attack. After all, even if there was a flaw, exploiting that flaw would mean developing an attack for that particular build of IOS. This was seen to be unlikely. In response, Recurity Labs' Felix Lindner, better known as FX, investigated just how different every version of IOS really was. What he found was a stable section, known as the boot loader, that could be reliably use to "initialize" code execution attacks against not just one version of IOS, but many.

Infrastructure was already under stress. The context of this paper is that we are at the cusp of a widespread deployment of very fast, very deeply inspecting systems. This paper shows how the *existing*, slow and somewhat lightly inspecting systems, nonetheless have quirks that by design and by implementation expose networks to risk. Code that is faster and more complex is not going to be any *more* secure; empirical evidence tells us the more code "in the line of fire", the more things can go wrong.

This is not an argument to not deploy systems. Obviously, there are many circumstances where the benefits from aggregating analysis of network traffic outweighs the risk of having that aggregator get exploited. Nobody should think that ALG's and Transparent Proxies should be decommissioned en masse. But prudence suggests that three steps would be good:

1. **Vendors** of network infrastructure, particularly infrastructure with DPI engines, should integrate code review and fuzzing into their testing operations. Products from Breakingpoint and Codenomicon, among others, make this sort of work operationally

viable. Furthermore, vendors should take further steps to make it easier and safer to quickly apply patches, perhaps by building rollback mechanisms or automated update for their devices. (In the case of the latter, vendors should be careful to acquire consent from administrators for retrieving new versions.)

2. **IT Purchasing** should evaluate systems based not only whether they meet immediate functional needs, but whether there's evidence that they've been tested. Perhaps a "proof of fuzzing" program could be developed, so as to provide competitive advantages to vendors who undertake this investment.
3. **IT Operations** should endeavor to apply patches, especially critical patches, to infrastructure with at least the same vigor that patches to Domain Controllers receive today. Obviously there are risks to patching – but as 2008 has shown us, there are risks not to.

B) Be *extraordinarily careful* about deploying active DPI that crosses organizational boundaries

In April of 2008, Jason Larsen and I discovered that ISP's around the world were deeply inspecting DNS traffic. In situations where they determined that a user was going to a nonexistent website, they'd insert advertisements rather than allowing an error page, or the browser's own advertising page, to render. This was done by actively transforming a NXDOMAIN (No Such Domain) response to a NOERROR response pointing to a set of ad servers. While we had no objection to nonexistent *domains* being hijacked, we quickly determined that nonexistent *subdomains* – such as http://nonexistent.google.com – were *also* being populated. While the content at those links simply immediately transitioned the user's browser to another domain, it would appear momentarily to a browser that a Google site was providing some Javascript to execute.

We quickly determined that it was straightforward to cause not simply the advertising provider's Javascript to execute, but our own as well. We could thus make Javascript appear to run within Google, within Live, within Yahoo, within Apple. We had subverted the entire web, at least the entire web behind infrastructure deployed at major ISPs.

The companies behind the infrastructure – the British firm, Barefruit, and the American firm, Paxfire – both fixed their security issues quickly. But the *exposure* remains – web sites around the world can actually be no more secure than these two company's infrastructure allow them to be – at least for users behind those ISPs.

DPI is just on the verge of being fast enough, and stable enough, to be put in front of fairly enormous network pipes – pipes that service not just one user, not just one organization, but entire regions and perhaps countries.

There is a *lot* of risk if things go wrong outside of organizational, and thus legal boundaries.

Passive DPI, which does nothing to alter data flows, should be mostly safe – perhaps there's an information disclosure risk, but little more. Active systems, by contrast, that are in a position to alter live traffic should be locked down as tightly as possible. Developers, and deployers should not think about what the system is *supposed* to do. Barefruit and Paxfire's ad server was *supposed* to just send users to an ad. Think about what they could do, in a worst case scenario, and attempt to engineer the DPI and the network to minimize damage.

## Technical Remediations

A) Whither Sockets?

Based on the findings of this paper, there will be some who call for the elimination of socket support from the browser. No sockets, and much of the ALG and fingerprinting attacks go away, as does the relevance of transparent proxy attacks.

I believe this is unfair – and not just because browser sockets have been around since 1996, long before firewalls came in.

The harsh reality is that networks are *stagnating* under a networking model in which if it doesn't look on the wire like the 2001 Internet, it doesn't work. It's not that communication isn't still happening – more than a few people have referred to HTTP as the "Universal Tunneling Protocol", as it gets through even when everything else is blocked. It's that the flexibility, and reliability of the systems we need to use are suffering under the constant need to tunnel everything over HTTP. When a single 1 byte keystroke, that could be transmitted with a single TCP packet, instead consumes an entire 1 kilobyte HTTP request, something is wrong.

This is even causing problems for security hardware. Consider the problem of ALG's. Much of the pain we have with them is that there are protocols that require multiple independent streams, each with their own error control and configuration. SCTP, a new reliable communication layer, offers exactly this. It also can't be deployed, because since it's not TCP or UDP, it won't get through firewalls. Firewall ALG's are, in a very real way, hoisted by their own petard.

Obviously, Java and Flash needed to fix their socket implementations, to not allow completely arbitrary socket communication to random internal and external addresses. But they're behaving as they should, only communicating with backends that opt-in. Is it possible that firewalls can play well with these browser sockets suppliers?

Originally, it was my belief that browser socket providers such as Java and Flash should implement stringent TCP and/or UDP port restrictions. After all, if all the interesting ALG's and proxies were on fixed ports, then a small range of ports would yield the minimum amount of vulnerability. Unfortunately, having spoken to firewall vendors, it is clear that their systems are all moving away from caring which port a given protocol is spoken on. Systems are fast enough now to handle ALG scanning on every possible port, and for functionality reasons, this has been somewhat necessary. So while Silverlight does in fact limit its sockets to a limited number of ports, in the long run, DPI engines will obviate this fix.

Furthermore, the reality is that browser sockets are often used to interact with systems that do operate on ports outside of the small ranges that Silverlight allows. Changing port policy now would break a lot of existing deployments, to a degree far worse than is generally tolerated.

So I've been forced to come up with something possibly better.

Flash supports two models for policy file retrieval. First, policies may be retrieved out of band. This means that, when Flash wants to know if it's allowed to generate a socket request to a given IP, it has to retrieve an XML policy *from that IP* declaring an appropriate allowing policy. If that fails, Flash can still try to retrieve a policy – however, it will issue the call on the IP and Port it's trying to speak to. If whatever TCP server replies with an accepting policy, then the connection will be torn down and a new one will be set up directly.

I propose a new mode: Upon receiving a certain "magic" reply, use the TCP session as is – expose it directly to the client. In this situation, we're basically building an ALG for Flash browser sockets, causing the policy returned by the server to have an additional element that says – "this is the firewall, please don't create a new socket, rather always retrieve policy and keep using this one".

Of course, when this element is included, all other ALG's would be disabled, suppressing both firewall bypass and fingerprinting. And since the transparent proxy wouldn't actually reply with an in-band policy, those attacks would be suppressed as well.

There are some problems. First, out-of-band policy retrieval would need to be suppressed, either by firewall or group policy. This would thus be mostly impossible for home routers to implement. Second, there are many situations where Flash is being hooked into a server that *cannot* have an inline policy server – for example, an IMAP server. This could potentially be dealt with by having the firewall itself convert the inline policy request it got from the client, into a standard out-of-band policy request using its own TCP/IP stack. Since sometimes Flash clients retrieve their policies from unusual ports, inline policy requests would need to be updated to include "hints" as to the real port from which to retrieve a policy.

The third problem is that there is no actual cross-plugin standard for how to authenticate browser sockets. Flash and Silverlight (at least for the sockets Silverlight allows) use vaguely similar policy files. Java, by contrast, uses reverse DNS, using simple enough packets that there's nothing obvious that could be firewalled in the first place.

Somehow, these various socket permissioning systems should be merged into one overall policy format. Such a thing would be useful not only for plugins, but for browsers themselves, which continue to struggle with establishing a coherent, cross-browser policy for access data across domain boundaries.

B) Transparent Proxy policy

As discussed earlier, transparent HTTP proxies have two choices when it comes to retrieving and caching content for a host that doesn't know the proxy is there:

1) It could recover the original IP address that the client was speaking to, and send a HTTP request to that IP address. It could then return Google's content.
2) It could ignore the original IP address, do a DNS lookup for www.google.com against its own name server, and return the Google content from the IP address returned

The first option reduces cache hit rates, potentially impacting performance and bandwidth costs severely. The second option breaks security, threatening click fraud and possibly internal network resources. Is there anything that can be done?

One possibility is to architect a transparent proxy solution that does not redirect traffic when the destination is to internal IP addresses. This can potentially be done within the redirection engines, or if that's not possible, by giving browsers explicit knowledge of the proxies they're supposed to use, along with a Proxy Auto-Configuration (PAC) script. PAC scripts provide the ability to use Javascript to evaluate each request against a set of rules, so as to determine whether the requests should go direct, or instead go through a proxy. Deploying PAC configurations can be tricky, but this is relatively easy to automate with either Group Policy or a technology called WPAD. Admittedly, this class of solution will not universally handle all web traffic like a fully transparent solution will, and it does require actual knowledge of internal IP space or at least domain names.

A superior approach was found in discussions with Robert Auger and Amit Klein. The concept we had was to implement lazy coalescing of cache content in a proxy server. In this approach, the first model (always retrieve traffic from the IP address the client originally used, and cache the retrieved content according to IP/Host) is followed. However, when convenient, the proxy server could complete a DNS resolution for the Hostname in question. *If* the IP address used by the client showed up as one of the IP addresses available for the Host, then the content would be cached against just the Host, rather than IP/Host. There'd be a somewhat reduced hitrate on a few highly volatile DNS names, but the vast majority of content would cache correctly once more.

There are two scenarios where the above breaks down. First, there is the situation identified by Adrian Chadd, who's one of the developers for Squid. Adrian noted that there might be situations where multiple proxies are chained in through one another – for example, a transparent proxy capturing all web traffic, forwarding everything through an anti-virus filter to prevent client infection. In such a situation, the second proxy would have no idea what the original destination address of the client was, so it would simply retrieve content from the Host declared in the HTTP header – thus, breaking Same Origin Policy.

Fixing this would require every proxy node to support something akin to an "X-Forwarding-For-Original" header, in which the transparent proxy would declare the actual destination address content is to be retrieved from, and the rest of the proxy chain respects that address as the ultimate destination. (The existing X-Forwarded-For header may be changed after the first in the proxy chain, it seems.) More details on this scenario can be seen in Robert Auger's paper, and in a soon-to-be-released document from Adrian Chadd.

C) Home routers

Regardless of whatever else is done, it remains a problem that web browsers can be DNS rebound into exposing the internal web interface of a home router to the Internet. It's perfectly reasonable that router manufacturers can't remove the default password. But perhaps the user could be cajoled into either pressing a button on the device, or power cycling it, in order to enable the default password? This would allow mostly the same functionality people are used to, without the widespread exposure of home router interfaces to remote compromise.

## Future Work

The final release of this paper will contain differentiators for many different devices, as well as a basic fingerprinter.

Passive FTP also has interesting firewall rules, that have been exploited through the years. A future version of this paper will discuss them.

Formal references will be added to this paper. It is assumed that, since its been some time since this ground has been tread, that there are more references to add. That's not a bug, that's a feature.

## Acknowledgements

Thanks to Robert Auger and Amit Klein for their extensive discussions on this class of issue, and to Microsoft, Sun, Adobe, Juniper, IBM, and Cisco for their help understanding the scope of this issue.