Hidden Internet connections

Part 2 - Intermezzo - The Internet

written by Steinowitz

November 1999

1 Before we start

I could have omitted this essay in this series, but it would have been wrong. Simply because I'd have to assume that you have a certain knowledge about the Internet which you may or may not have. Therefore, to make sure that you have this knowledge about the Internet, I'll write a whole essay on this subject.

Essays somehow related to this one are 'Reversing website visitor information' and 'Gaining access to secured website', which I've both written in November 1999. You can find them on the DREADed website. (The URL is at the bottom of this document.) And, of course, there's much more information on this subject.

2 From IP address to protocol

We all know that the Internet is nothing but an immense number of computers all connected to eachother. That's not really true. The most important about the Internet is that *all* computers are able to communicate with *all* other computers connected to the Internet. It doesn't matter which processor architecture is used, it doesn't matter what operating system is running, they're all able to communicate!

You can't achieve this by just connecting some computers, you need a set of very good 'communication rules' which all computers know and obey. If, and only if, this is the case, you can create a network as large as the Internet.

Why all these stupid thoughts, you may wonder. But I don't write them down without reason. I want you to learn more about the 'communication rules' of the Internet. When you get to know them, it becomes much easier to reverse programs dealing with Internet connections.

Everything in this essay will be software-oriented: sometimes, it may seem to you as if we are dealing with hardware items. (For example, you may already have seen the header 'Ports'.) But that won't be the case. We aren't dealing with hardware, we aren't dealing with physical connections. We use them, that's all.

2.1 IP address

2.1.1 Unique address

It would be rather useless if we would connect all those computers using the networks if we wouldn't be able to for those computers to communicate with others. But how do we achieve this? We would need to assign a unique address to each computer! That's exactly what an IP address is: a unique address each computer connected to the Internet has.

An IP address looks like this:

xxx.xxx.xxx.xxx where *xxx* is a number in the range 0-255

The only exception on this is the first number: it must be at least 1. It's no coincedence that this range is from 0 to 255: each number of an IP address could be coded with 8 bits. The complete IP address could therefore be coded in only 32 bits!

One computer may have multiple IP addresses. Especially for large servers, this can be useful. On the contrary, multiple computers may not have the same IP address! Otherwise, the whole idea of a unique address would be gone!

You now know in which ranges the numbers of an IP address have to be. How about calculating the maximum number of computers which could be (directly) connected to the Internet? (I must say 'directly', because there are ways to access the Internet without having an IP address. More on this later.)

2.1.2 Dynamic and static IP addresses

You now know the number of computers which could be connected to the Internet is not infinite. An Internet provider (ISP) can't say: well, we're out of IP addresses right now, but we'll think of some new ones. That's impossible. Companies, universities and ISP's can request a certain number of IP addresses and it's possible that a certain range of IP addresses is then assigned to the requester. Once assigned, only the association the IP addresses are assigned to are allowed to use these (until they declare that they don't need them anymore).

What does an ISP when it has more clients with dial up accounts than IP addresses? The ISP uses *dynamic* IP addresses. Each time a client dials up, the dial up server assigns one of the available IP addresses to this computer. And when the computer hangs up, the IP address is ready for use by someone else.

There are also many computers which have a *static* IP address, which means that this computer always has the same IP address when it's online. Webservers (which are online all day long) should have static IP addresses. How could we reach them otherwise? We originally wanted to assign IP addresses to make sure that we would have a unique addresses we could use to reach other computers!

2.2 Hostnames and DNS

It would be rather difficult for us if we would have to remember the IP addresses of all computers we would like to get information from. How about surfing the web, for example. Would you like surfing to *212.23.34.45* instead of *www.target.com*? I know what I like better! Imagine how all advertisements

would change - no longer the easy-to-rememer *www.company.com*, but only difficult IP addresses...

Now that we have a (numerical) unique address for each computer connected to the Internet, it's getting time to register names for our computers. When you have a computer with a static IP address, you can register a *domain name*. A domain consists of a name and a country code: *cjb.net*, *search.nl*, *yahoo.com*, etc.

When you register such a domain name (which is not free!), you have to state the IP address of your computer. You also need a DNS service. *DNS* stands for *Domain Name Service*. This Internet service makes it possible to lookup the IP address of a computer when you have a hostname. It's also possible to lookup the hostname when you have an IP address.

For example, when you tell your browser to go to *www.target.com*, your browser asks a DNS server what the IP address of *www.target.com* is. The browser then contacts the target computer using the give IP address and requests the files of the website at that webserver.

Once you have a domain name and a computer with an IP address the domain name 'redirects' to, it's possible for you to create 'subdomains'. *subdomain.target.com* is a subdomain of *target.com*. The large benefit of these subdomains is that you can do several things with them: you could assign subdomains to computers with different IP addresses than the 'main' computer, but you could also use the subdomain to decide which website on the 'main' computer the visitor wants to visit.

If you have a normal dial up account and your ISP uses dynamic IP addresses, you'll probably have a hostname like *98acab27.ipt.aol.com* while your online. (Keep in mind that your IP address changes every time you dial up and that your hostname therefore also changes a little!) As you see, there may be subdomains of subdomains: *98acab27.ipt.aol.com* is a subdomain of *ipt.aol.com*. On the other hand, if your ISP uses static IP addresses, your hostname while you're online will probably look like *username.isp.com*.

One last remark: an IP address has to be a unique address for only one single computer. A hostname is unique, but it's not necessarily assigned to only one computer.

2.3 Ports

We know what an IP address is, we know what a hostname is, but we still have a little problem when we want to connect to our remote host. Many servers provide many different services at the same time: FTP, WWW, Telnet, SSH, time, etc. How does a server know what we want when we connect? A server wouldn't understand this: 'I'd like to get file X from the rootdirectory. I'm sorry, but I don't know if it must be file X in the FTP root or file X in the WWW root.'

This wouldn't work. We need a way to distinguish between FTP and WWW. Especially because each Internet service is provided by a different program and a remote host can only connect to one of these programs.

There is a very nice solution to this: each computer connected to the Internet has a large number of (virtual!) *ports*. To be more accurate: each computer has 65,536 ports (could be coded with only 16 bits!). **IS THIS CORRECT? I DON'T REMEMBER EXACTLY**

Whenever we connect to a remote host, we connect to a certain port. We don't see this very often, but it's true. When we don't specify a port, our browser will automatically connect to port 80, because that's the port for the WWW. We could specify a port if we would need to: *www.target.com:300*. Our browser would lookup the IP address belonging to this hostname and it would subsequently try to connect port 300 of that computer.

We'll probably get an error message if we do this: the webserver software isn't *listening* to port 300, it's only listening to port 80! I just introduced a new term: a program can *listen* to a certain port. Client programs want to connect to other computers and connect to a certain port. Server programs, on the other hand, are listening to one or more port(s) and can accept any connections from outside. A port may not be used by more than one program at the same time.

2.4 Protocol

2.4.1 Computers trying to communicate

We are (finally!) able to connect to our target host: we know a hostname, we have a DNS server which we can use to lookup the IP address belonging to this hostname and we also know the port we should connect to. Is there anything which could stop us from communicating with other computers? Yes! Let me give you a conversation between two computers.

```
<YourPC> Hi there!
<RemoteMac> Hey, I'm a Mac webserver, what can I do for you?
<YourPC> Umm, could you please do PC_UAHDNFDAOEKFMSJEKXKW?
<RemoteMac> I'm sorry, but I don't understand you, I'm a MacIntosh.
<YourPC> GPF, WINDOWS CRASH! (Expected to get PC_UAHDNFDAOEKFMSJEKXKW,
not this.)
```

Of course, computers don't communicate like this, but I think you get the point. We need a good set of rules which both computers obey to make sure that they'll understand eachother and won't be surprised about certain replies.

Since each Internet service is a little different and requires slighlty different commands and communication, one set of rules wouldn't be enough. Therefore, we have a lot of standardized sets of communication rules: *protocols*.

A protocol defines how data is transfered, how certain data should be requested and many more things like this. When your browser connects to a webserver at port 80, a connection between your browser and the webserver software is established. These two can now communicate and exchange data, because they both use exactly the same protocol: *HTTP (HyperText Transfer Protocol)*.

Server software and client software can now communicate without problems: they use the same protocol, they know what they should expect. This way, we avoid conversations like the one I wrote down above.

There are many different protocols. HTTP is used for the World Wide Web, *FTP (File Transfer Protocol)* is used to transfer files to or from a remote host, *Telnet* is used for controlling remote computers you have access to and so on and on. All these protocols are standardized to make sure that all software producers use exactly the same version of the protocol. These standardized protocols are

defined in RFC's, so if you want to learn more about specific protocols, you should search for these RFC's on the Web.

2.4.2 Server software, protocols and ports

When we were discussing the IP addresses, I already mentioned that we don't want to remember numbers. We want easy-to-remember names! And hostnames are very suitable for that. But we also need a port number! It'd be very annoying if we would have to remember both a hostname and a port number.

Therefore, protocols (and thus Internet services) are linked with the port numbers. The standard port for HTTP is 80, the port for FTP is 21, the port for Telnet is 23, the port for IRC is 6667, etc. This way, a program like a browser knows that it should connect to port 80 if you want to visit a website. On the other hand, if you want to access an FTP server (you enter *ftp://ftp.target.com*, where *ftp://* means that FTP is the protocol which should be used), your browser automatically connects to port 21.

There are, of course, exceptions: it's very easy to configure your server software a little different to make it listen to a port different from the standard.

3 Sockets

We have all information we need to know to be able to connect to a server program running at a remote computer, but how do we connect to it? As you see, we are making progress with every section and subsection, however, new questions rise just as fast...

Fortunately, we don't have to write any device drives or anything like that: we don't deal with hardware stuff like sending commands to a modem. All operating systems I know have a hardware abstraction layer which does that for us. All we have to know is how this abstraction layer works.

When you want to establish a (virtual) connection to a remote computer, you have to use *sockets*. Let me tell you a little more about sockets.

3.1 Berkeley sockets

During the beginning of the Internet, two API's were developed for accessing a TCP/IP-based network like the Internet: Berkeley sockets and the UNIX System V Transport Layer Interface. Both of these were developed in C on UNIX systems.

Berkeley sockets became the most-used API: TLI is still available under UNIX operating systems, but other operating systems don't support them. Berkeley sockets are supported by almost any system, which makes it easier to port applications. (However, Microsoft did the same with Berkeley sockets as they did with Java, as we'll see later in this essay.)

The Berkeley sockets API provides everything you need for (virtual) Internet connections: you can connect to remote hosts, you can send and receive data, you can lookup IP addresses and so on and on.

As you can imagine, this API works with 'sockets', but what exactly is a 'socket'? It's hard to describe, because it's a very abstract thing. You could say it's a phone, but I won't proceed with that comparison with that anymore, you've probably had enough of that.

For each Internet connection with a remote computer, you need a separate socket. A program can ask the API for a socket with certain capabilities: you could create a socket listening to connections from other computers òr you could create a socket you want to use to connect a remote computer. When a program exits, the sockets it started are gone. You tell your sockets what they should do and they'll do it.

3.2 Windows sockets

Winsock (Windows Sockets) is an API based on the original Berkeley sockets, though Microsoft changed it a little. Compare it with Java: Sun produces an open source coding language with open source Virtual Machines. And what does Microsoft? It makes its own version of Java which isn't 100

Even before Java existed, Microsoft already adapted the Berkeley sockets and changed them a little. The most important changes were a couple of new functions which should be called to initiate Winsock and to clean things up.

The changes Microsoft made make it more difficult to write portable Internet applications, although there are ways to reduce this to a minimum. In C, for example, you could use preprocessor directives to choose between UNIX and Windows source code:

```
#ifdef UNIX
#include <arpa/inet.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#else
#include <winsock.h>
#endif
```

If you don't fully understand this: doesn't matter, I only mention it for the coders who would like to know. Much more about Windows sockets in the next part of this series.

4 Last words

That was it for this essay. In the next part, I'll explain more about the Winsock API and DLL's and I'll also show you how to deal with WinTECH's SocketSpy/32. That will be the last part of the introduction to this series. After that, we'll have the knowledge we need to start with and we'll start with the real reversing work. The first target I'll write about (in part 4) is HotDog Professional 5.5.

Last, but not least, personal greetz go out to everyone I know... Special greetz go out to ˜S˜, you did a great job!

Steinowitz
switz@newmail.net
http://dread99.cjb.net and *http://www.s.url*