OnionCat – A Tor-based Anonymous VPN

Bernhard R. Fischer

2048R/5C5FFD47 <*bf@abenteuerland.at*>

December 18, 2008

Abstract

Tor is an anonymizing network. It allows users to anonymously access Internet services. Its architecture guarantees that the real IP of users cannot be revealed in any way. Tor also provides so-called *Hidden Services*. Those are services which are hidden within the Tor network. This means that not only the user stays anonymous but also the service (destination). Hidden services have several benefits but unfortunately they are not very user-friendly and they have some protocol restrictions.

OnionCat manages to build a complete IP transparent VPN based on those hidden services, provides a simple well-known interface and has the potential to create an **anonymous global network** which could evolve to a feature- and information-rich network like we know the plain Internet today.

1 Introduction

Tor is an anonymizing network[1] within the Internet consisting of several nodes capable of forwarding TCP/IP sessions through it thereby hiding the origin at the destination end point. The location of a user i.e. his IP address is hidden at the remote site, e.g. the IP address of a user accessing a web service will not be revealed in the server's log files. Instead the IP of a random Tor exit node appears. An exit node is a Tor node at which a TCP/IP session leaves the Tor network.

This is a great feature because it improves a person's privacy especially if somebody resides under aggravating circumstances. Unfortunately Tor is not only used in the "right manner". Some one could also missuse it and if done right nobody will every discover who did wrong because even for the Tor network itself it's impossible to reveal the originating IP – deliberately it's a design feature. Always only the IP of the exit node appears in the public and depending on the law of the country where the exit node resides in it could lead to a law-enforced service shutdown or even something worse.

That's why people are usually not willing to run exit nodes.[2]

1.1 Hidden Services

The counterpart of a user who would like to hide his location is a service which should be hidden, i.e. a service which you know that it exists and you know how to access it but you don't know where it is. Basically it could be any type of service, e.g. a web service.

In plain old Internet this is more or less impossible because an IP address can always be traced back to an Internet provider and finally to a user or company. Hidden services [2] are services which exist only within the Tor network and of course they are also location hidden. That means they are not identified by an IP address but by an .onion-URL and the Tor network is able to find the right path to it but neither the user nor the Tor network can detect the IP address.[3]

Beside location hiding there is a second great benefit: connections to hidden services do not

[1] See the Tor project page www.torproject.org for general information.

[2] That's not the only reason but probably the most important one.

[3] Of course only if the service is configured correctly.

leave the Tor network. No single exit node is needed and that's perfect because, as already mentioned, exit nodes are rare and because of that they are permanently traffic overloaded which results in a high latency.

Another benefit is that Tor guarantees end-to-end encryption from the client to the hidden service which is not true for connections to the Internet even when using Tor.[4]

That's why the use of hidden services is really interesting. Providing them increases the usability of Tor and the privacy of users and service providers.

1.2 The Problems

Unfortunately these .onion-URLs look like random numbers and characters – and in fact they are more or less random – which makes them really hard to remember, even harder than IP addresses because they have 16 digits.

But who really needs to remember IP addresses? Everybody uses names today. There is the domain name system (DNS) which resolves names to IP addresses. In plain Internet, name service is one of the most important ones. Nearly every user and every service uses names instead of IP addresses while using the network. The introduction of DNS – a distributed name resolution service – made the Internet more usable and opened it to a wider community.

But within Tor there is currently no resolving mechanism available for translation of names to .onion-URLs. Traditional DNS can not be used that easy because it is IP-based[5] (specifically the Internet class IN) and hidden services are .onion-URL based which can not be simply exchanged with IPs. From the Tor point of view those URLs are already names. Theoretically, an approach could be to use canonical names (CNAME) pointing to .onion-URLs but this would break authentication. Unlike IP addresses .onion-URLs provide authentication, i.e. using the .onion-URL a user can verify that a service really is the right hidden service and not any other one who pretends to be the right service. DNS basically does not interact with services that are associated with names, i.e. it cannot provide authentication as it is used for Tor and the security of users and services.

Even if someone deals with those .onion-URLs it's still not easy to use hidden services because the interface between an application and Tor is SOCKS [5] – a protocol for proxying TCP/IP. From a software modularity point-of-view it is a good idea to use SOCKS because it is a well standardized interface and many applications support it. But many do not! And every application that supports it needs user interaction to setup the right settings for SOCKS. A user should be able to **use hidden services without any differences to regular Internet services.**

Furthermore SOCKS version 4 only supports TCP/IP. There's no transport for UDP and other layer 4 protocols. Typically DNS is based on UDP which is an important protocol but cannot be used in combination with Tor.

2 Basic Considerations

Based on the previously mentioned considerations we suggest an **application interface on the IP layer**. With such an interface every protocol based on IP should be transportable.

On most operating systems such interfaces are available and provided by the kernel. On Linux, *BSD and other Unices there are kernel modules providing a layer 3 tunnel interface, usually called *TUN* device[6] but it's also available on Darwin (probably because of its BSD code base) and a similar model is available even on Windows.

OnionCat shall connect only to hidden services. As already mentioned they are addressed by .onion-URLs which are requested through SOCKS4a [4] and resolved by Tor itself. Obviously, because .onion is not a valid top level domain (tld) in Internet DNS.

Unfortunately, if using layer 3 which usually is IP, there's no such thing like a host name. We need a new IP-compatible addressing scheme for hidden services but this cannot be done by just setting up a DNS service which resolves .onion names. It would break the authentication

[4] Unfortunately many users do not know this fact and believe that everything gets encrypted just because they use Tor.

[5] That's not a matter of design but a matter of the real (IP) world.

[6] It's similar to the TAP device which is a layer 2 interface.

scheme of Tor's hidden services and it would imply user interaction again to configure a specific DNS that hosts the .onion tld.

Tor generates a .onion-URL [6] out of the public key of hidden services. It's exactly an 80 bit wide Base32 encoded string. Those 80 bits are one half of the SHA-1 hash of the public key. And that of course is derived by the private key. That's why those URLs are strongly related to the hidden service. We do not want to loose any of those bits because it would increase the probability for collision attacks thereby breaking the authentication scheme again[7] and it would deny reversability.

7 bits	1	40 bits	16 bits	64 bits
Prefix	L	Global ID	Subnet	Interface ID

Figure 1: Unique-local address format.

We use IPv6 addresses as a new addressing scheme for hidden services. IPv6 addresses are 128 bit wide, that's large enough for including 80 bits of an .onion-URL. According to RFC5156 [1] we use a network out of the *unique-local* address space. These are reserved for internal use in networks comparable to those of RFC1918 [7] of IPv4. As shown in Figure 1 the basic address format has a fixed minimum prefix length of at least 48 bits, additionally variable 16 bits for subnetting and 64 bits for the interface ID (host part). We don't need any subnet so we add the full subnet part to the interface part resulting in an 80 bits wide host part. The prefix length for those addresses is 48 bits.

48 bits	80 bits
FD87:D87E:EB43	.onion-URL

Figure 2: OnionCat addressing scheme.

According to RFC4193 [3] we set the "L"-bit to 1 and generated a global ID thus resulting in the new unique-local IPv6 prefix FD87:D87E:EB43::/48 – the OnionCat prefix. Address translation is easy by Base32-decoding the .onion-URL and inserting those 80 bits into the host part of the IPv6 address (see Figure 2). E.g. decoding 7fd22jhmqgfl45j6.onion leads to 0xf947ad24ec818abe753e. Putting this together with our OnionCat prefix it results in the IPv6 address fd87:d87e:eb43:f947:ad24:ec81:8abe:753e.

[7] With time it's getting even worse because there are known collisions in SHA-1 yet. [9]

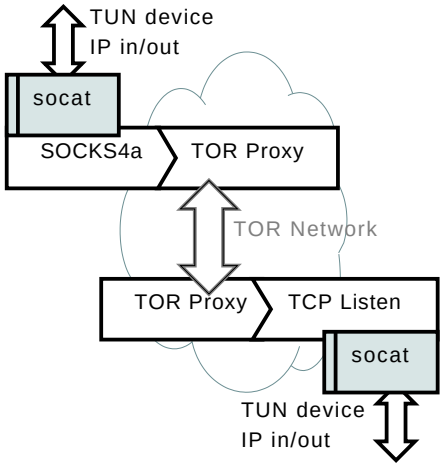

Figure 3: Socat hidden service connection.

Based on this addressing scheme we can now translate .onion-URLs to IPv6 addresses and vice versa.

OnionCat basically works similar to socat [8]. Socat[8] is a relay that handles all kinds of streams that are associate with a *file descriptor* (...in Unix everything is a file ;-). It has two ends each associated with a file descriptor and forwards data between them. For our application specifically interesting is the feature to forward data between a TUN device on one end and a SOCKS4a connection at the other end and at the opposite a TCP listener on one end and a TUN device again on the other end respectively (Figure 3). Now you could assign IP addresses to both ends and your IP-transparent point-to-point connection is ready. This setup has to be done manually.

Different to socat OnionCat **automatical connects through Tor** based on the .onion-URL related IPv6 addresses and it is able to build up **point-to-multipoint connections** because of its routing capability (Figure 4). The appropriate IPv6 address is assigned automatical to the TUN device which creates an entry to the kernel's IPv6 routing table. Hence, packets are forwarded to OnionCat by the kernel without further interventions.

[8] It's also father of the name "OnionCat".

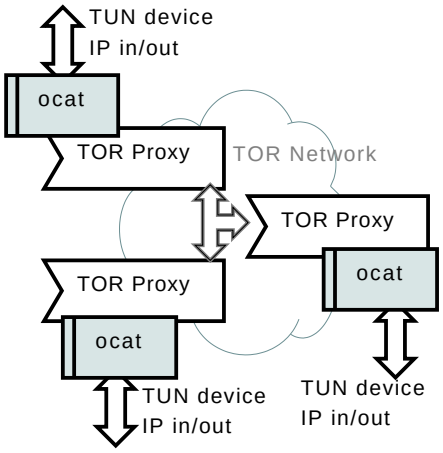

Figure 4: OnionCat connections.

3 OnionCat Implementation

OnionCat is a multi-threaded application which basically receives packets on the TUN device and forwards them across the SOCKS4a connection through Tor and vice versa, once connections to remote OnionCat hidden services are established. Internally it maintains a *peer list*. Every *peer* is associated with an .onion-URL and its appropriate IPv6 address, the file descriptor of the TCP session (between OnionCat and Tor), an idle time value, some counters, and a defragmentation buffer.

Periodically the *Socket Cleaner* thread (see Figure 5) checks the idle times of the peers. If one exceeds the limit it is removed from the list and the Socket Receiver (see Section 3.2 below) thread is signalled that the peer list has changed.

The actions taken are different if packets are received either through the TUN device (outbound direction from a local point of view) or through a TCP session between OnionCat and the Tor proxy (inbound direction).

3.1 Outbound Direction

Packet reception on the TUN device is handled by the *TUN Receiver* thread (Figure 5). It extracts the destination IPv6 address of the incoming packet and *looks up* wether a peer with this address exists in the peer list or not. If so, it forwards the packet directly to the peer's file descriptor, updates his idle timer and continues receiving packets on the TUN device.

If there's no peer in the peer list it *initiates a new connection* by triggering a sleeping *SOCKS Connector* thread. The packet itself is queued for a while and gets forwarded by the *Packet Dequeuer* thread[9] after the peer is ready. Directly after the packet is queued the TUN Receiver continues receiving packets on the TUN device.

The SOCKS Connector spawns a new spare thread and tries to *connect* to the hidden service through the Tor proxy's SOCKS4a interface. It blocks until the connection is established by Tor. If it was successful it makes a new entry into the peer list, signals the Socket Receiver that the peer list has changed and terminates. If the connection failed it just terminates thereby dropping the request. The SOCKS Connector threads maintain a list of new peer requests.

3.2 Inbound Direction

Data reception from Tor is done by the *Socket Receiver* thread if connections are already established. If data is received it is appended to the defragmentation buffer of the appropriate peer. Every peer has its own defragmentation buffer. If the buffer contains at least one complete packet the source IPv6 address is extracted from the header and copied into this peer's address field if it is still empty (It will be explained shortly why this could occur). Then it forwards the packet to the TUN device and deletes it from the defragmentation buffer.

New incoming hidden service connections from Tor are handled by the *Socket Acceptor* thread. On program startup it creates a listening TCP socket and waits for connections (currently on default port 8060). Once a connection comes in it accepts it, creates a new entry in the peer list and continues accepting connections. The Socket Receiver is signalled that the peer list has changed.

At this time it does not know about the originating address (.onion-URL/IPv6) because those TCP sessions are always initiated by the local Tor proxy, hence, its source address is 127.0.0.1 (or ::1). Furthermore, it is just the transport, OnionCat (and every other hidden service) just uses the payload of those session. Outbound packets cannot be sent to this new peer as long as it is not identified. Identification happens im-

[9] This thread is *not* depicted in Figure 5.

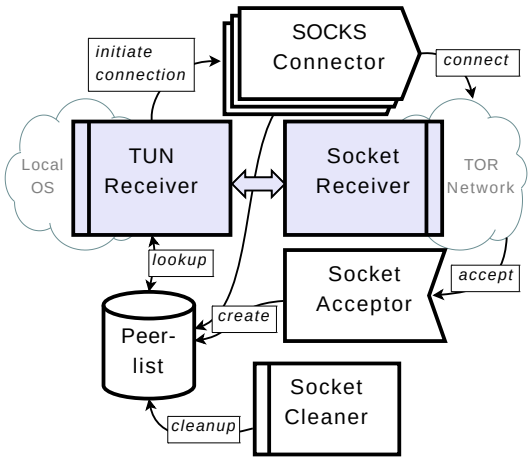

Figure 5: Internal structure.

mediately at reception of the first IPv6 packet (see above).[10]

3.3 OnionCat and IPv4

At a first glance it looks easy to do IP[11] forwarding if IPv6 does already work. But as mentioned in Section 2 hidden services are addressed by an 80 bits wide ID which we managed to convert to IPv6 addresses. Unfortunately that's not possible with IP addresses because they're only 32 bits wide. We could strip some bits off the .onion-URL and pack it into an IP address but sadly this type of conversion isn't reversible, but that's a requirement.

OnionCat does IP forwarding with a routing table which represents the glue between an IP address and an IPv6 address (and an .onion-URL respectively). Currently we use the least significant 24 bits of the IPv6 address and put them into the host part of the private network 10.0.0.0/8. The last 24 bits of the address fd87:d87e:eb43:f947:ad24:ec81:8abe:753e are 0xbe753e. Translating this to IP using our method results in 10.190.117.62. This IP address together with the netmask 255.0.0.0 is also assigned to the TUN device and an entry in the kernel routing table appears.[12] All packets with a destination therein will travel across the TUN device to the TUN Receiver thread (Figure 5).

The problem that now occurs is that the peer cannot be looked up in the peer list because the destination IP address cannot be reversed to an .onion-URL. As already mentioned, OnionCat maintains a second list which is a routing table with destination IP addresses, netmasks, and appropriate IPv6 gateways. On reception of an IP packet the TUN Receiver looks up an entry in the routing table and then further looks up the gateway address in the peer list and continuous as described in Section 3.1 or drops the packet if no routing entry exists. This routing table has to be setup manually.

4 Availability and Application

The source code of OnionCat can be downloaded at the current project home page www.abenteuerland.at/onioncat/. It is released under the GNU GPLv3 and is in *early* development state at the time of writing this paper. Currently it runs under Linux Kernels 2.6.x and 2.4.x, FreeBSD 6.x, OpenBSD 4.x, and Mac OS X 10.4 and 10.5, but maybe also under other operating systems. It's written portable as possible. The most ugly part is porting the TUN device initialization code (congratulations to the OpenVPN guys!).

Have a look at our project page for a description on OnionCat usage. We do not maintain a mailinglist yet, but we plan to do so. Announcements are currently done on the or-talk list.

The goal of what OnionCat is made for is to recreate the Internet on an anonymous basis: *AnoNet*. If everybody – this includes users and service providers, i.e. people, organizations, companies, etc., providing services – uses Tor and OnionCat, this could become reality.

For now it may be used for smaller user groups which need to exchange data basically with the same requirements as those using Tor but in a more transparent or more flexible way in respect to the underlying network protocols. People could setup private meeting rooms, chat relays, or similar services.

Of course, there are also dark sides. With OnionCat people can also do e.g. file sharing completely anonymous. But maybe this has a

[10] This is a known security weakness but we'll find a solution.

[11] Subsequently I'll use the term IP instead of IPv4.

[12] Obviously, it may overlap other routing entries with subnets of 10.0.0.0/8 but OnionCat currently is in early development state and we don't care at the moment. We'll make this configurable in the future.

good side effect: if file sharing is done only within Tor, the exit nodes will become less overloaded.

5 Conclusion

In this paper we presented a method for making Tor's hidden services more user-friendly and transparent. This is done by insertion of OnionCat, a layer between client applications and the Tor proxy thereby lowering the access layer from TCP to IP. This change in layers also forces another addressing method for which we showed a deterministic reversible approach. By acting on the IP layer every protocol beside TCP can be transmitted without further circumstances across Tor.

OnionCat creates the major advantage of **using Tor's hidden services like usual IP hosts** on the Internet. Together with Tor, it has what it takes to build *AnoNet* – a perfect **anonymous Internet** within the Internet. This creates the interesting problem of anonymous not back-trackable payment methods.

During development and test phase we discovered some problems. OnionCat currently lacks authentication on incoming connections. It just uses the first incoming packet for identification.

IPv4 support is not very mature. OnionCat maintains an own routing table. It would be more comfortable if it shares the kernel routing table but we think that this might include portability issues.

The connection setup to a hidden service could last very long. Sometimes this takes one to two minutes. With this kind of transparency that OnionCat creates, also RTT measurements are easy, e.g. with the ping command in the simplest case. We observed RTTs between 1 and 30 seconds in the real Tor network[13] and unfortunately this seems not to be a matter of OnionCat.

Another problem seems to be that Tor is based on TCP. The circuits are built of concatenated TCP sessions. If creating TCP sessions through OnionCat it leads to "TCP-over-(TCP+TCP+...+TCP)". TCP uses algorithms to dynamically adapt to bandwith availability and this stack of dynamic systems could lead to very ugly behavior. Packet transmission sometimes looks like they are travelling through rubber bands which means that they are delivered in periodical occurring bulks.

But beside all that problems, we still believe in OnionCat and Tor and we think that this new kind of **anonymous VPN is a great benefit** for the people on this world.

[13] We maintain a private Tor network in our lab for test purposes.

References

[1] M. Blanchet. Special-Use IPv6 Addresses. RFC 5156 (Informational), April 2008.

[2] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The Second-Generation Onion Router. https://www.torproject.org/doc/

[3] R. Hinden and B. Haberman. Unique Local IPv6 Unicast Addresses. RFC 4193 (Proposed Standard), October 2005.

[4] Ying-Da Lee. SOCKS 4A: A Simple Extension to SOCKS 4 Protocol. http://ftp.icm.edu.pl/packages/socks/-socks4/SOCKS4A.protocol.

[5] Ying-Da Lee. SOCKS: A protocol for TCP proxy across firewalls. http://ftp.icm.edu.pl/packages/socks/-socks4/SOCKS4.protocol.

[6] The Tor Project. Tor Rendezvous Specification. http://www.torproject.org/svn/-trunk/doc/spec/rend-spec.txt, 2008.

[7] Y. Rekhter, B. Moskowitz, D. Karrenberg, G. J. de Groot, and E. Lear. Address Allocation for Private Internets. RFC 1918 (Best Current Practice), February 1996.

[8] Gerhard Rieger. socat - Multipurpose relay. http://www.dest-unreach.org/socat/, 2007.

[9] Xiaoyun Wang, Yiqun Lisa Yin, and Hongbo Yu. Finding Collisions in the Full SHA-1. http://people.csail.mit.edu/yiqun/-SHA1AttackProceedingVersion.pdf, 2005.