Tunneling simplified (XFLTReaT framework) Revision 1

Balazs Bucsay

@xoreipeip
Hack In The Box GSEC
25. August 2017

Abstract. This paper aims to recognize the similarities between existing tunneling solutions and gives advice on possible framework implementation. The reference implementation can be found on Github under the name of XFLTReaT. With this framework it is possible to use only one tunneling program to use different transport protocols to tunnel data. This approach can help on both sides of the IT-security industry to implement new attack and defense scenarios.

Table of Contents

Introduction

Tunnels and VPNs are with us for a long time; these solutions are used in our daily life, sometimes even without our knowledge. Around 2000, the Universal TUN/TAP device driver was implemented in different Unices (Linux(1), FreeBSD, Solaris) that made it easier to create tunnels and use them as transport channels between endpoints. These drivers helped developers to create programs that can be paired up with virtual interfaces to handle packets that are going to or coming from the kernel. No hardware had to be installed to emulate these kinds of interfaces and it became easier to develop tunneling solutions.

One of the most famous open-source VPN is the OpenVPN(2) that can utilize both TUN and TAP drivers. This tool is widely known and used by companies, professionals and end-users. While OpenVPN only supports TCP and UDP as transport protocols (and also has support for HTTP proxies over HTTP CONNECT, which is essentially just TCP with a little overhead at the connection phase), but there is no support for other protocols that are lower or higher on the OSI layers.

A number of tools are already created for tunneling over other protocols that are situated on lower or higher layers of the OSI model, for example for ICMP there are the icmptunnel(3), icmptx(4), Hans(5), etc. DNS that is located on the application layer tgat can also be used for tunneling. Iodine(6) is one of the most famous tools that is used to exploit this property of the protocol.

Any other protocol that has a payload section and is capable to transport data from A to B can be used for tunneling. This paper tries to fill the gap that is present in this field for many years by recognizing the similarity between existing solutions and the need for a universal implementation.

1 Tunneling

1.1 Tunneling basics

The easiest way to understand how tunnels work is through Virtual Private Networks (VPNs) as they are widely used nowadays A number of reasons why they are widely used is listed below:

- Accessing the internal network of the company when working remotely
- Hide the real IP address
  - ◇ Journalists to communicate anonymously
  - ◇ Whistleblowers
  - ◇ Torrent usage
  - ◇ Etc.
- Bypass ISP related filtering (NetBIOS, SMTP, website blacklisting)
- Bypass captive portals
  - ◇ Airports, cafés
  - ◇ Guest networks

All VPN solutions are composed of two things:

- VPN server/concentrator
- VPN client

To create a VPN connection, the following steps are usually done:

- Connection created between the two endpoints (client and server)
- Authentication and key exchange
- Virtual interface setup on both sides with private IP addresses
- Routing set up table setup on client
- Data exchange started

By taking these steps, the client creates a tunnel (or bridge) between the server and the network behind that (which can be a private network of a company or the Internet itself) and all the traffic is sent to the server over the tunnel.

Depending on the configuration, the routing can also be set up to create a split tunnel. In case we are dealing with a split tunnel configuration, only some routes will be added to the routing table and only packets addressed to these IP ranges will be sent to the server over the tunnel while the default route stays the same. Split tunnels are out of scope of this paper.

The main difference between browsing the Internet without a tunnel (Fig. 1) and going over a tunnel (Fig. 2) is that all of our traffic goes to the server first, that forwards the packets to the original destination; this means that the route of the packets was changed.

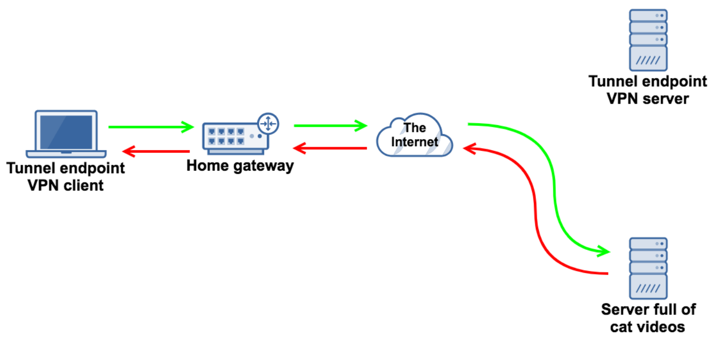

Fig. 1. No tunnel set up

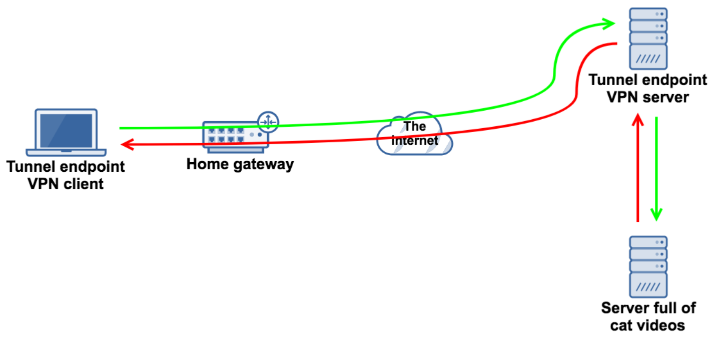

Fig. 2. Tunnel built and in use

If the VPN solution supports encryption, then all the packets are encrypted between the client and server, so that gives an additional layer of security. As routing have changed by creating the tunnel and sending everything over that, (in most of the cases) the overall number of hops will increase, the latency will grow and part of the route will be always the same (the route until the packet reaches the VPN server). This also means that the addressed destinations will only see the VPN server's IP address that forwarded the packets from the client.

Advantages of the VPN solutions:

- Gives additional security by encrypting the traffic
- Hides the real IP address

Disadvantages:

- Increased latency (for most of the times)
- Longer routes
- Reduced throughput (MTU)

1.2 Tunneling 101

Strictly speaking about tunneling all the solutions are the same, they choose a transport protocol that is capable of transmitting data and connecting two endpoints together. The only real difference is how the data is handled and how the packets are encapsulated in the transport protocol.

In modern operating systems, the drivers or modules (TUN and TAP) are capable of setting up virtual interfaces that can act as real interfaces. From the user space there is no difference between a virtual and real device, both can be configured in the same way.

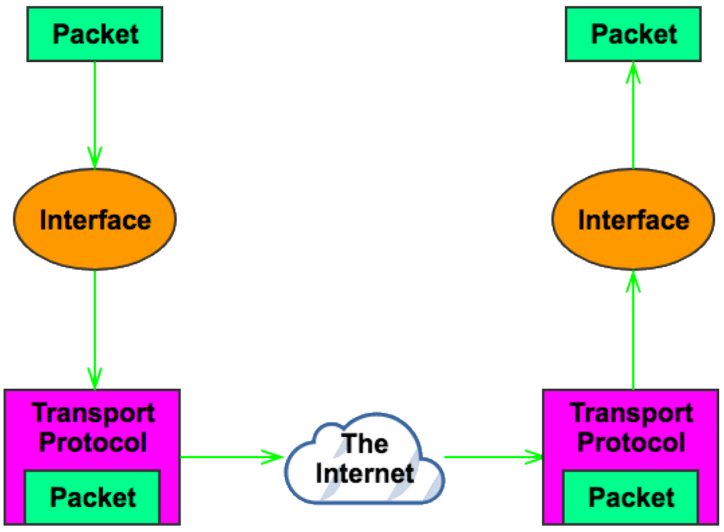

Fig. 3. Tunneling simplified

When a tunneling program starts, it sets up a virtual interface that acts as a network card and any packet sent to this interface is handled by the program. Tunneling tools are responsible for encapsulating packets in a way that can be sent over the network to the server. The server does the same in the reverse direction. When an encapsulated packet is received over the network, it gets decapsulated and then sent to the previously set up virtual interface.

If a protocol was designed to be capable of transmitting data (there is a payload section in it), then it can be used for tunneling. A number of examples for such protocols are TCP, UDP, ICMP, DNS, HTTP, SMS (GSM).

On networks where everything is filtered but ICMP packets, ICMP tunneling is a great way to bypass the restrictions and get unfiltered Internet access. All packets have to be encapsulated in ICMP packets and sent to the server. Figure 4 shows a UDP packet that cannot be sent over the network because of the firewall rules in place. However, if that packet is encapsulated in an ICMP packet as indicated on figure 5, it could be sent to the other endpoint.

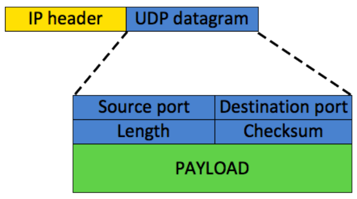

Fig. 4. Original UDP packet

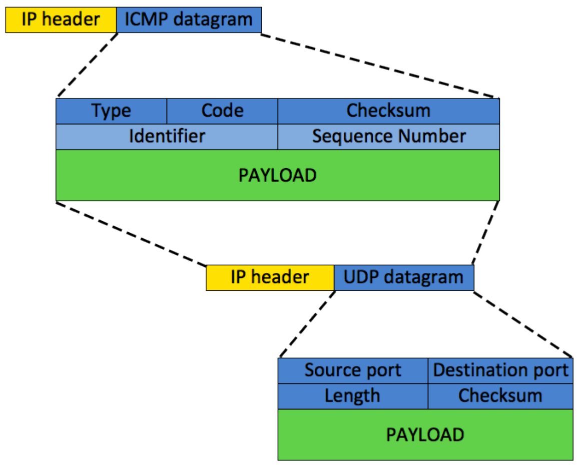

Fig. 5. ICMP encapsulated UDP packet

The server's responsibility is to decapsulate the original packet from the transport protocol packet and to write it to the tunnel interface. From that point, the kernel will handle the forwarding and other necessary things.

1.3 Maximum Transmission Unit

The Maximum Transmission Unit or MTU is the maximum size of a packet that can be transmitted over the network. This value is usually assigned to an interface as a property. If the packet size including the IP header is over this limit, then the kernel or the network device is responsible for fragmenting the packet to create smaller packets that are just equal size or smaller than the MTU. After fragmentation, the packet is split into multiple chunks and sent over the network. The kernel or the network device at the recipient is then responsible to reassemble the original package. The default MTU on most network devices and interfaces is set to 1500 bytes, so all the packets have to fit in 1500 bytes including headers.

From tunneling perspective this is an important property. Encapsulating packets introduces an overhead by placing the original packet into another payload section. As a result this increases the size of the new packet. If the original packet was 1500 bytes long and it gets encapsulated into a different packet, the size will grow over the 1500 bytes. If this happens it will either be fragmented into two packets or network devices will reject it. The easiest way to solve this problem is to decrease the MTU value of the local interface to make sure that smaller packets will be sent.

In certain cases there is a need to send encapsulated packets in other encapsulated packets or let us say tunneling over a tunnel. By looking at the Matryoshka doll or Russian nesting doll (Fig. 6) it is easy to understand how the MTU works. The MTU has to be decreased each time the packet is encapsulated because of the overhead, otherwise it does not fit into the other, bigger MTU size.

Fig. 6. Matryoshka doll, decreseasing MTU

Let us see how the fragmentation works. The theoretical size limit of a UDP packet is 65535 bytes, but in reality it is only 65507 because of the limitation imposed by the IPv4 protocol (the IP header is 20 bytes and the UDP header is 8 bytes).

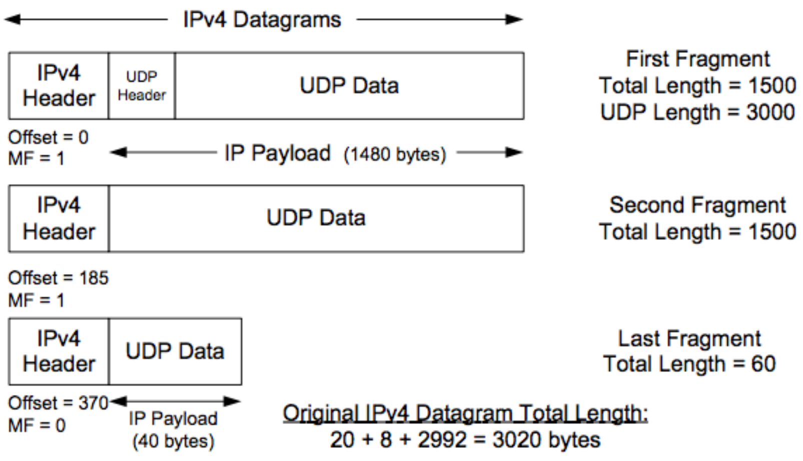

Fig. 7. UDP fragmentation (7)

If a UDP packet has to be sent over the network and that is bigger than the MTU, then it has to be fragmented. As an example take a look at (Fig. 7) where the payload is 2992 bytes long, with the UDP header it adds up to 3000 bytes and the IP header would increase 20 bytes more. In total, the full package is 3020 bytes long, which is obviously bigger than the default MTU, which is 1500 bytes. For that reason this packet has to be fragmented into 3 different packets. The first will be 20 + 8 + 1472 = 1500 bytes; the second will not need the UDP header since it is fragmented (20 + 1480 = 1500 bytes) and the rest, which is 20 + 40 = 60 bytes long. In total 3060 bytes in 3 packets will be transmitted over the network instead of one packet that is 3020 bytes long.

1.4 TUN and TAP

Most modern operating systems support two different types of virtual network devices, the TUN and TAP; both of them can be used to create one or more virtual interfaces. These are only virtual, so any packet that is sent to them will not go beyond the kernel. While the TUN devices work at layer three or IP level in the OSI reference model and only IP packets can be sent to them, TAP devices work at layer two at the Ethernet level. TAP devices can be used for bridging and are usually used in virtualization systems. In case of tunneling solutions the IP level or layer three is just enough, so there is no need to build or amend Ethernet frames and IP headers. If the IP forwarding is enabled in the operating system and the proper firewall rules are set then the entire low-level networking (for example: fragmentation or packet readdressing for forwarding) is gone, these are all handled by the device or the kernel.

2 Framework

2.1 Problems with existing solutions

Several tunneling implementations can be found on the Internet for multiple protocols. Although there are some decent implementations that are still maintained, in great majority the solutions are not more than a proof of concept or end of life codes.

Problems in general:

- EoL/PoC codes:
  - ◇ If a bug is found, that cannot be reported, since there is noone who will fix that.
  - ◇ Codes do not follow changes in the operating systems.
  - ◇ In order to add new functionality the user needs to touch the code.
- Different programming languages are used most of the times:
  - ◇ Lack of modularity, parts of the code cannot be reused elsewhere.
  - ◇ Lack of portability; the codes are running on one operating system only.
- The configuration files totally differ from solution to solutions:
  - ◇ The user has to find out how and what to modify.
- No documentations or how-to-s.
- As many protocol as many solutions. Implementations have support for only one or two transport protocols.
- Tools do not make it easy for the users to map out the network weaknesses; lack of automation in most of the tools.

These problems are very generic in the field of IT-Security. Most security professionals are not coders, therefore they are just creating proof of concepts to show the world that their ideas could work. Unfortunately, until this point there were no attempts to reform the field of tunneling, this paper aims to give guidance and a reference implementation of a potential framework that can solve these problems.

2.2 Requirements for a framework

The framework that can solve the above mentioned issues could be implemented by following these requirements:

- Open-source
  - ◇ Community work always produce great tools, Linux and Metasploit framework are just two great examples for this
  - ◇ Being feature rich and having good ideas implemented
- Easy to use and understand programming language
  - ◇ Script languages are usually easier to use than compiled languages
  - ◇ Automatic line indenting helps
  - ◇ Widespread or hyped languages always help on community development.
- Modularity
  - ◇ Handling different modules for different goals.
  - ◇ Transport protocol modules have to implement the basic properties of the protocol and how that handles the data.
  - ◇ Authentication modules have to implement different authentication methods to authenticate clients.
  - ◇ Encryption modules have to implement different encryption methods that can be used to encrypt the data flow.
  - ◇ The use of modules has to be possible in a plug and play fashion. Only the configuration file has to be modified to enable or to use a module.
- Multi client support
  - ◇ Clients have to be handled from the framework, should not be handled from the transport protocol modules, except a few cases and methods.
  - ◇ Authentication, encryption and module specific user properties have be stored in the client object that is handled by the framework
  - ◇ The client object has to be extendable
- Object Oriented
  - ◇ Possibly to use Object Oriented Programming (OOP) for implementation
  - ◇ Transport modules should be built upon their parent protocols (e.g. SOCKS Proxy on TCP or DNS on UDP)
- Check functionality
  - ◇ All modules have to have a check function that send a challenge or challenges to the server to solve. If the challenge is solved, the tunnel can be built.
  - ◇ This functionality helps users to check connectivity and makes low-level network mapping unnecessary or at least less necessary
- Ease of use and development
  - ◇ User friendly
  - ◇ Should be easy to develop plugins for it

As mentioned earlier, this paper only gives recommendations about how a potential framework can be created and a reference implementation in Python called XFLTReaT follows this paper, that can be accessed from these URLs:

`http://xfltreat.info/`
`https://github.com/earthquake/XFLTReaT/`

2.3 Interface

The framework have to create its own interface, it can be either TUN or TAP. Since the majority of the Internet protocols are based on IP, there is no real need for TAP, although there could be cases where TAP is a must.

By creating an interface the framework should use the configuration to set the properties (IP address, netmask, MTU, etc.). There is no need to set up a new interface for every module, one should be more than enough. However this means that all traffic directed to the users will come from the interface and have to be selected and directed to the right client. A packet selector module should be created to handle this problem.

When a transport protocol module receives an encapsulated packet from the client or the server, it decapsulates it and writes on the interface. The kernel will change the IP header along with other necessary modifications. It either changes the destination IP address to the client's private address or the source address to the server's IP address. The modified packet can then be sent to the original destination. If the direction is client to server, then the story is pretty straightforward, however in the opposite direction it is a bit harder. The only way to know where the packet should be sent to, is the private IP address in the IP header. The framework on the server side must have an internal database of the clients and all clients must have a writeable pipe. This pipe will replace the tunnel interface from client point of view when it comes to read.

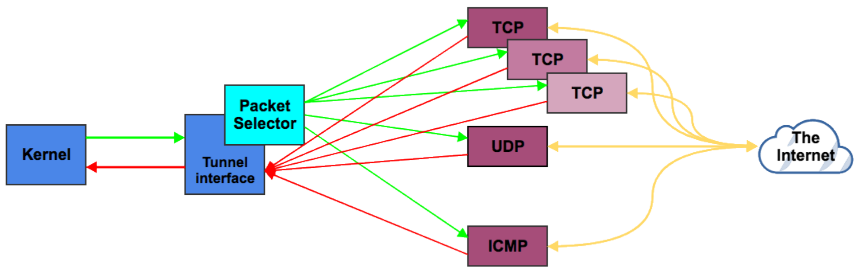

Fig. 8. Data flow

Figure 8 clearly shows how the server side should operate. The clients connect from the Internet directly to the modules. Modules can be anything as far as the framework supports that transport protocol module. If the client was using TCP tunneling, then all data will be sent to the server over TCP (as indicated by the yellow lines). Then the TCP module decapsulates the packet, writes it to the Tunnel Interface (red lines), which was set up when the framework started. The kernel makes the necessary changes on the packet and forwards it to the original destination. If there is an answer from the original destination, that will be sent to the server, this then will be forwarded to the Tunnel Interface (green lines). The Tunnel Interface or the kernel will amend the packet again and forwards it to the client.

The Packet Selector takes place here, it reads all the incoming packets from the interface and tries to match to the private IPs from the internal client database. If a match was found, then the packet will be written on that client's pipe (as indicated by the green lines). The TCP module will look for a change on that pipe, if that happens it reads the packet and sends back to the client (over the yellow line) after encapsulation. From the module point of view the entire packet selection appears to be transparent, it is just the same as it was reading from interface itself.

2.4 Routing

After the interface was properly configured, the client must set up the routing. First, the original default route has to be determined and saved, then that has to be removed from the routing table (step 1). A new default route should be added with the destination of the server's private address (step 2). This step ensures that all packets arrive to the virtual interface first. One last rule has to be added that will allow the communication with the framework's server, this rule should strictly set the destination as the IP of the server (which can be an intermediate one in case of DNS tunneling or proxies) and the gateway to the IP that was the original default gateway (step 3).

When the client terminates, the original routing table has to be reverted. The default route has to be replaced (step 2) with the original default route (step 1) and the second rule, which was added, has to be removed as well (destination: framework's IP, gateway: original default gateway). No other modifications are necessary, since this is not a split tunnel.

2.5 Multi client support

The framework has to serve multiple clients at the same time. When a client connects to the server, the framework has to create a client object that has to store at least two properties, the public and private IP of the client. Different modules can have different requirements based on how they work, this needs to be acknowledged and the client object has to be extendable because of this. The extendibility should be done by inheritance, as this provides the best way to store other properties and to be compatible with the framework. If the transport protocol module requires different attributes, then the client object has to be recreated from a support file (see later).

The client object has to have different methods to get the details of the client, including the private and public IP addresses as well as the writeable pipe that stores the packets coming from the interface. Having these methods implemented, the Packet Selector can handle the incoming traffic from the server side and can forward it to the right client.

2.6 Transport Protocol Modules

These modules are the heart and soul of the framework. The whole purpose of the framework is to make the development of the transport protocol modules easier than it was previously. This can be done by realizing that all the existing solutions are based on the same principal and by finding the common points that are not varying between protocols. If all of these could be implemented in a framework and the modules could be just built up with these functions, then only the differences have to be implemented.

By having that said, the transport protocol modules should only implement how the data is sent and received. This varies among protocols, because the encapsulating method always changes, sometimes it is needed to set certain flags or to build up special headers and so on.

If there are some requirements that the framework cannot fulfill, that has to be implemented into the module or the framework has to be improved.

2.7 Support files

There should be some logical boundaries introduced when coding the modules. When the transport protocol module needs to use some protocol related implementations (for example ICMP header builder, ICMP checksum, DNS query builder, etc.) that should be placed into different, so called support files (as they are supporting the module). The modules should be clean, tidy, lightweight and well structured without the parts that could be reused in other modules.

If more general functions are needed for the module that are not present in the framework, those functions should be moved to the core in order to be reusable in future modules.

2.8 Check Functionality

All transport protocol modules should have one or more check functions implemented. The goal of this functionality is to help users find out which communication channel can be used for tunneling on an unknown or even on a known network. This avoids unnecessary manual low-level network mappings. Properly written check functions could replace the need for tools like Wireshark(8) or Nmap(9) to save time.

The client has to send a challenge to the server. This challenge does not have to be hard or cryptographically secure, the point of this message is to see whether the server can get this message or not. If the challenge reaches the server, then it answers to the client with the solution. When the client receives a correct solution, we can come to the following conclusions:

- There is a server on the other side that is using the framework
- The communication channel works and a tunnel can be created

2.9 Auto-tune

Multiple auto-tune functions should be implemented in certain modules, for example DNS where different DNS server implementations can behave differently. These checks can be used for auto-tuning the tunnel or just error debugging. A few examples that can be checked in case of DNS:

- Rate limitation
- Maximum length of a query
- Maximum length of an answer
- Record type support (NULL, PRIVATE, TXT, CNAME, MX, etc.)
- Encoding support (base128, base91, base64, base32)

2.10 Control and Data channels

The framework needs to be capable of using two different virtual channels, the data and control channels. These are communication channels over the actual tunnel between the client and the server. The data channel is only used to transfer data between the two endpoints, and the control channel is used to exchange control messages such as:

- Authentication related messages
- Key-exchange for encryption
- Check functionality
- Logoff message
- Keep-alive messages
- Error correction messages
- Congestion control

2.11 Module tree

There are many similarities between transport protocol modules, which should not be re-implemented each time a new module is created. If object orientated programming is used, then every new module can inherit the methods and the properties from its parent, therefore unnecessary code reuse can be circumvented. As an example, take a look at figure 9.

Generic module should have all the methods (prototypes) and properties that will be used across the modules. Because the stateful and stateless connections have to be handled in a different way, there must be a split in the module tree there. These three modules will be referenced as skeleton modules in the following as they give a skeleton for the rest of the other modules. For stateful modules (like TCP or HTTP) all common methods and properties have to be implemented in the Stateful module. The Stateless module has to be created likewise. As an example, SOCKS only have to implement how the client connects to the SOCKS proxy, when the connection was made and when the stream was created between the client and the server over the proxy, the same methods could be used, as it was a simple TCP connection. There is no need to implement the send and receive functions, handle the control and data messages because all of these are already implemented in the parent class, in TCP.

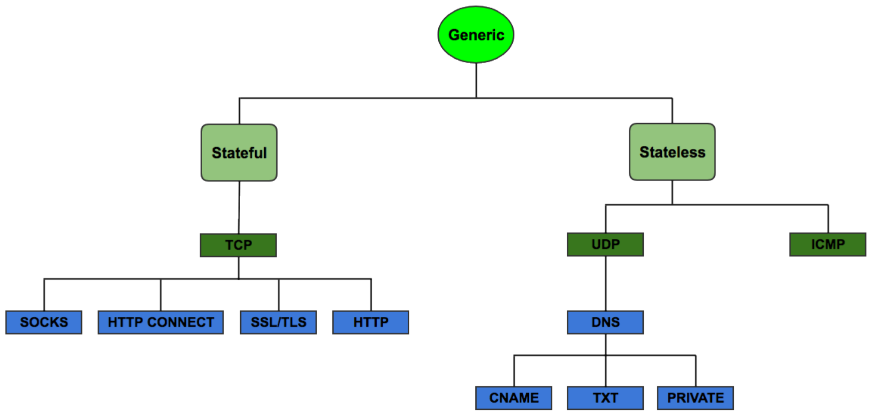

Fig. 9. Module structure tree

2.12 Stateful and Stateless connections

There is a big difference between stateful and stateless connections. In general, stateless connections are built on UDP or ICMP, while stateful connections are built on TCP. When a TCP connection is made, the associated socket receives packets only from that connection, but this does not hold true for stateless connections. In that case everything that was sent on the port or on that protocol will arrive to that socket. Just like in the Packet Selector when the server has to find out which client sent the message and make sure that:

- The client is a valid client
- The client is authenticated
- Protocol related properties are saved to the client object (for example sequence number for ICMP)

All modules should run in a different thread, if the connection is stateful, then all client or connection could run in a different thread as well. This could provide a better stability; in case of an unexpected error only the client or the module will be affected instead of the whole framework.

2.13 Transport Protocol Module implementation guidelines

If all commonly used functions in the transport protocol modules were implemented in the framework, then only the rest have to go into the modules. Although these functions will differ between modules, but the high level functionality will be the same. The functions described below have to be implemented in all modules and that can be done either by inheritance or implementation.

Send function This function will be responsible for encapsulating messages and for sending them to the server or to the client on the protocol that the module supports. For example, if this is an ICMP module, then all the support functionalities like ICMP header builder or checksum is implemented in the support library. These functions will be used in the send function to create a valid ICMP packet with the message inside. Before the message is encapsulated, encoding or encryption can be used on the raw message.

Receive function The same as the send function but in reverse order. This receives the packet from the socket and parses it. The message from the packet can be transformed; it should be decoded or decrypted before returning the raw message. Other necessary values should be extracted from the packet and saved for future use. For example, in case of ICMP, the identifier and sequence number are needed to bypass the Network Address Translation (NAT) and firewalls.

Communication initialization function Before executing the of the communication part, some variables may have to be initialized or certain functions have to be called. This function should initialize all the necessary settings before the core of the module starts.

Communication function This function should handle the socket and file descriptors, when new data appears on any of those, it should be read and handled. The socket could be either the socket of the connection with the server or the socket of the client that is connected to the server. One of the file descriptors can be the tunnel's file descriptor (in the case of the client side) or the file descriptors of the pipes that are associated with the authenticated users. These pipes are created by the Packet Selector, to replace the need for direct interface access and have distinct direct access to different client streams.

Serve function The serve function will be called on the server side only. When the module wants to work as a server, this function will be called. This has to set up a socket, bind to it and listen for connections. Necessary variables have to be initialized here too. When these are all done, the communication initialization and communication function have to be called.

Connect function The connect function works similarly as the serve function, but on the client side. A socket for the connection has to be set up then the connection has to be made. If the connection requires some protocol specific things (like HTTP CONNECT requires an initial CONNECT request to create the TCP stream), then that has to be handled here. After establishing connection, the authentication phase should start by calling the authenticate function. If the authentication was successful the communication function should be called.

Check function The client and check functions are almost the same. The only difference is that in case of the check function, instead of calling the authenticate function, it iterates over the check functions to see which approach could be used to build up a tunnel.

Configuration sanity check function All modules require different configuration settings, which should be checked before accessed or used. This function has to be responsible for sanity checks and error messaging. If a property is missing or the value of that is not well formed, then an error message should be printed on the screen to inform the user about the error. The modules only have to check their own settings to ensure that no malicious or erroneous values will be used during the execution.

Intermediate hop function If the module uses an intermediate hop to access the framework's server (for example for DNS that would be the DNS server or for SOCKS that is the proxy server) then this function has to return the IP address of that intermediate hop. This should be called when the routing is set up to make sure that the intermediate hop can be addressed directly.

Authenticate function Based on the configuration, the proper authentication function has to be found and called. This should be a wrapper on the authentication modules, since this function decides which module is called.

Check function The check function only needs to check whether it is possible to build up a channel between the two endpoints. If the challenge arrives to the server and the correct result was sent back, then the check function should return true, which means that the connection can be built. Otherwise it should return false.

Auto-tune functions Auto-tune functions will differ from module to module. Certain modules such as TCP do not require more than one message exchanged as the transportation channel is always the same. In other modules like ICMP it is common that network devices maximize the payload length. Some firewalls does not allow bigger ICMP packets than a static value, so this value have to be mapped out with multiple auto-tune functions to reach the maximum throughput. The same applies to DNS, where many things can change. Some DNS servers do not support the theoretical limit for record name length or strictly follow the RFC, which does not distinguish between lower- and uppercase letters. These differences can again be mapped out with several probes. If the appropriate way was found to reach the maximum throughput on that specific network then the auto-tune functions return with the best variant.

Cleanup function This function has to be responsible for cleanup after the module exits. All sockets and file descriptors should be closed, except the interface's file descriptor as this will be closed by the framework upon exit. All other variables must be deinitialized and the memory must be freed.

3 Usage

This framework can be used for both offensive and defensive purposes.

3.1 Offense

The most trivial reason to use this tool for is to bypass different obstacles. If the network was firewalled, but only a port or one protocol is allowed, then it can be configured to use that port or protocol for tunneling. The server can be set up to bypass that filtering and the client can gain unfiltered Internet access or even exfiltrate data. The possibilities are endless in this field. The challenging part is to find firewall misconfigurations, but the check functionality can help with this. If the misconfigurations are more advanced, then low-level manual checks have to be done and the framework has to be adjusted accordingly. In case the firewall is properly set up, but there is an internal proxy that forwards only special requests, then the HTTP or Proxy modules have to be changed to support this kind of communication (like Kerberos authentication or a special header in the HTTP request).

The main objectives for offensive security are the following:

- Get unfiltered Internet access
- Exfiltrate data

Both of these can be achieved with this framework.

3.2 Defense

As usual, the defense is more challenging and more complex than offense, because on this side they do not only have to deal with the technology, but with the business too. Any modifications on the network or appliances, modifications on the accustomed settings in a company could have a business impact. Just like in offense, this framework can be useful for defense as well. The server should be set up outside the organization and should be configured according to the organization's setup. If there is an HTTP proxy in use, then HTTP module should be configured; if the proxy supports the CONNECT method, but only on port 443/tcp, then the HTTP CONNECT module should listen on port 443/tcp and so on. The testers should know about the organization's configurations and its weaknesses to exploit those. The check functionality can help to check whether it is possible to bypass the internal policies and protections.

It is very important to mention that companies should try to use the framework before attackers exfiltrate data. This is to see how to adjust their protective mechanisms to avoid future exfiltration attacks that are based on these protocols. With the exfiltration, not only the internal policies and protection mechanisms can be tested, but the SOC teams as well.

4 Mitigations

4.1 Captive portals

Some organizations are maintaining networks that are utilizing captive portals. This technology is more than usual nowadays, and the main reason for using these is to control access to a network or gather information about the users. In the majority of the cases, captive portals and the surrounding configurations are misconfigured and there are more than one way to bypass them. Captive portal solutions should be configured as detailed below. Until the client has not authenticated himself or herself on the captive portal:

- All external and internal directed traffic should be filtered. All packets should be dropped or redirected to the captive portal (except those that are addressing the portal itself).
- Inter-client communication should be dropped as well (all time).
- Only A (ipv4) and AAAA (ipv6) record DNS requests should be answered
- All DNS requests should be rewritten to the captive portal's IP address

These points ensure that the connected client cannot communicate with anyone but the captive portal. After authentication all traffic can be allowed trough.

4.2 DNS tunneling

Fighting against DNS tunneling in a company is not an easy task. If the DNS server is not available, most of the services and internal processes could stop, making mistakes in this field could cause outage. Probably one of the best ways to mitigate this kind of tunneling is the following:

- All traffic that is going to or coming from the Internet should be filtered
- Only the HTTP proxy should have Internet access
- HTTP Proxy should be enforced on all computers including servers and workstations
- HTTP Proxy should do the DNS resolving instead of the client
- External DNS names should NOT be resolved by internal hosts
- Internal DNS server should resolve only internal addresses
- There will be exceptions in all companies, those set of machines should be handled differently
- Have a separate DNS server that resolves external addresses for the exceptions

Organizations should include these points into their planning phase before building up their network, otherwise it could be difficult to amend the existing architecture.

5 Conclusion

The paper made an abstract of all tunneling protocols and identified the similar and identical parts among them. It provided a possible solution on how to implement a proper framework that handles all the similarities and differences with minimal code duplication.

The attached reference implementation proves the paper, shows that it is possible to simplify the tunneling process and make it universal between transport protocols. Until now, different transport protocols had to be used for different solutions. This paper and the reference implementation aim to change this and try to help both sides of the IT-security community to recognize the potential in this field again.

The reference implementation is still under development, but it can be accessed on the following URLs:

`http://xfltreat.info/`
`https://github.com/earthquake/XFLTReaT/`

References

[1] Maxim Krasnyansky, Maksim Yevmenkin and Florian Thiel. Universal TUN/TAP device driver. `https://www.kernel.org/doc/Documentation/networking/tuntap.txt`

[2] OpenVPN Inc.. OpenVPN - Open Source VPN `https://openvpn.net/`

[3] Dhaval Kapil. icmptunnel `https://dhavalkapil.com/icmptunnel/`

[4] Edi, Siim Põder, Thomer Gil. ICMPTX `https://github.com/jakkarth/icmptx/`

[5] Friedrich Schöller. Hans `http://code.gerade.org/hans/`

[6] Erik Ekman and Bjorn Andersson. iodine `http://code.kryo.se/iodine/`

[7] Shichao's Notes `https://notes.shichao.io/`

[8] Gerald Combs. Wireshark `https://www.wireshark.org/`

[9] Insecure.Com LLC. Nmap `https://nmap.org/`