

Product Solutions Resources Open Source Enterprise Pricing

Search

Sign in

Sign up

yarrick / iodine

Public

Notifications

Fork 502

Star 6.2k

<> Code

Issues 7

Pull requests 6

Actions

Security

Insights

master

<> Code

934 Commits

.github/workflows

doc

man

src

tests

.gitignore

.vimrc

CHANGELOG

LICENSE

Makefile

README-android.txt

README-win32.txt

README.md

Readme

ISC license

Activity

6.2k stars

130 watching

502 forks

Report repository

Releases

14 tags

Contributors 32

+ 18 contributors

Languages

C 97.3%

Makefile 2.3%

Other 0.4%

README

ISC license

iodine - https://code.kryo.se/iodine

This is a piece of software that lets you tunnel IPv4 data through a DNS server. This can be usable in different situations where internet access is firewalled, but DNS queries are allowed.

COMPILING

Iodine has no configure script. There are two optional features for Linux (SELinux and systemd support) that will be enabled automatically if the relevant header files are found in /usr/include . (See script at ./src/osflags)

Run make to compile the server and client binaries. Run make install to copy binaries and manpage to the destination directory. Run make test to compile and run the unit tests. (Requires the check library)

QUICKSTART

Try it out within your own LAN! Follow these simple steps:

On your server, run: ./iodined -f 10.0.0.1 test.com . If you already use the 10.0.0.0 network, use another internal net like 172.16.0.0 .

- Enter a password.
- On the client, run: `./iodine -f -r 192.168.0.1 test.com` . Replace `192.168.0.1` with your server's ip address.
- Enter the same password.
- Now the client has the tunnel ip `10.0.0.2` and the server has `10.0.0.1` .
- Try pinging each other through the tunnel.
- Done! :)

To actually use it through a relaying nameserver, see below.

HOW TO USE

Note: server and client are required to speak the exact same protocol. In most cases, this means running the same iodine version. Unfortunately, implementing backward and forward protocol compatibility is usually not feasible.

Server side

To use this tunnel, you need control over a real domain (like `mydomain.com`), and a server with a public IP address to run `iodined` on. If this server already runs a DNS program, change its listening port and then use `iodined` 's `-b` option to let `iodined` forward the DNS requests. (Note that this procedure is not advised in production environments, because `iodined` 's DNS forwarding is not completely transparent, for example zone transfers will not work.) Alternatively you can forward the subdomain from your DNS server to `iodined` which must then run on a different port (`-p`).

Then, delegate a subdomain (say, `t1.mydomain.com`) to the iodined server. If you use BIND for your domain, add two lines like these to the zone file:

```
t1      IN      NS      t1ns.mydomain.com.      ; no
t1ns    IN      A       10.15.213.99
```

The `NS` line is all that's needed to route queries for the `t1` subdomain to the `t1ns` server. We use a short name for the subdomain, to keep as much space as possible available for the data traffic. At the end of the `NS` line is the name of your `iodined` server. This can be any name, pointing anywhere, but in this case it's easily kept in the same zone file. It must be a name (not an IP address), and that name itself must have an `A` record (not a `CNAME`).

If your `iodined` server has a dynamic IP, use a dynamic DNS provider. Simply point the `NS` line to it, and leave the `A` line out:

```
t1      IN      NS      myname.mydyndnsprovider.com.      ; no
```

Then reload or restart your nameserver program. Now any DNS queries for domains ending in `t1.mydomain.com` will be sent to your `iodined` server.

Finally start `iodined` on your server. The first argument is the IP address inside the tunnel, which can be from any range that you don't use yet (for example `192.168.99.1`), and the second argument is the assigned domain (in this case `t1.mydomain.com`). Using the `-f` option will keep iodined running in the foreground, which helps when testing. iodined will open a virtual interface ("tun device"), and will also start listening for DNS queries on UDP port 53. Either enter a password on the commandline (`-P pass`) or after the server has started. Now everything is ready for the client.

If there is a chance you'll be using an iodine tunnel from unexpected environments, start `iodined` with a `-c` option. Resulting commandline in this example situation:

```
./iodined -f -c -P secretpassword 192.168.99.1 t1.mydomain.com
```

Client side

All the setup is done, just start `iodine` . It takes one or two arguments, the first is the local relaying DNS server (optional) and the second is the domain you used (`t1.mydomain.com`). If you don't specify the first argument, the system's current DNS setting will be consulted.

If DNS queries are allowed to any computer, you can directly give the `iodined` server's address as first argument (in the example: `t1ns.mydomain.com` or `10.15.213.99`). In that case, it may also happen that *any* traffic is allowed to the DNS port (53 UDP) of any computer. Iodine will detect this, and switch to raw UDP tunneling if possible. To force DNS tunneling in any case, use the `-r` option (especially useful when testing within your own network).

The client's tunnel interface will get an IP close to the server's (in this case `192.168.99.2` or `.3` etc.) and a suitable MTU. Enter the same password as on the server either as commandline option or after the client has started. Using the `-f` option will keep the iodine client running in the foreground.

Resulting commandline in this example situation, adding `-r` forces DNS tunneling even if raw UDP tunneling would be possible:

```
./iodine -f -P secretpassword t1.mydomain.com
```

From either side, you should now be able to ping the IP address on the other end of the tunnel. In this case, `ping 192.168.99.1` from the iodine client, and `192.168.99.2` from the iodine server.

MISC. INFO

IPv6

The data inside the tunnel is IPv4 only.

The server listens to both IPv4 and IPv6 for incoming requests by default. Use options `-4` or `-6` to only listen on one protocol. Raw mode will be attempted on the same protocol as used for the login.

The client can use IPv4 or IPv6 nameservers to connect to `iodined`. The relay nameservers will translate between protocols automatically if needed. Use options `-4` or `-6` to force the client to use a specific IP version for its DNS queries.

If your server is listening on IPv6 and is reachable, add an AAAA record for it to your DNS setup. Extending the example above would look like this:

t1	IN	NS	t1ns.mydomain.com.	; no
t1ns	IN	A	10.15.213.99	
t1ns	IN	AAAA	2001:db8::1001:99	

Routing

It is possible to route all traffic through the DNS tunnel. To do this, first add a host route to the nameserver used by iodine over the wired/wireless interface with the default gateway as gateway. Then replace the default gateway with the `iodined` server's IP address inside the DNS tunnel, and configure the server to do NAT.

However, note that the tunneled data traffic is not encrypted at all, and can be read and changed by external parties relatively easily. For maximum security, run a VPN through the DNS tunnel (=double tunneling), or use secure shell (SSH) access, possibly with port forwarding. The latter can also be used for web browsing, when you run a web proxy (for example Privoxy) on your server.

Testing

The `iodined` server replies to `NS` requests sent for subdomains of the tunnel domain. If your iodined subdomain is `t1.mydomain.com`, send a `NS` request for `foo123.t1.mydomain.com` to see if the delegation works. `dig` is a good tool for this:

```
% dig -t NS foo123.t1.mydomain.com
ns.io.citronna.de.
```

Also, the iodined server will answer requests starting with 'z' for any of the supported request types, for example:

```
dig -t TXT z456.t1.mydomain.com
dig -t SRV z456.t1.mydomain.com
dig -t CNAME z456.t1.mydomain.com
```

The reply should look like garbled text in all these cases.

Mac OS X

On Mac OS X 10.6 and later, iodine supports the native utun devices built into the OS - use `-d utunX`.

Operational info

The DNS-response fragment size is normally autoprobeed to get maximum bandwidth. To force a specific value (and speed things up), use the `-m` option.

The DNS hostnames are normally used up to their maximum length, 255 characters. Some DNS relays have been found that answer full-length queries rather unreliably, giving widely varying (and mostly very bad) results of the fragment size autoprobe on repeated tries. In these cases, use the `-M` switch to reduce the DNS hostname length to, for example 200 characters, which makes these DNS relays much more stable. This is also useful on some “de-optimizing” DNS relays that stuff the response with two full copies of the query, leaving very little space for downstream data (also not capable of EDNS0). The `-M` switch can trade some upstream bandwidth for downstream bandwidth. Note that the minimum `-M` value is about 100, since the protocol can split packets (1200 bytes max) in only 16 fragments, requiring at least 75 real data bytes per fragment.

The upstream data is sent gzipped encoded with Base32; or Base64 if the relay server supports mixed case and `+` in domain names; or Base64u if `_` is supported instead; or Base128 if high-byte-value characters are supported. This upstream encoding is autodetected. The DNS protocol allows one query per packet, and one query can be max 256 chars. Each domain name part can be max 63 chars. So your domain name and subdomain should be as short as possible to allow maximum upstream throughput.

Several DNS request types are supported, with the `NULL` and `PRIVATE` types expected to provide the largest downstream bandwidth. The `PRIVATE` type uses value 65399 in the private-use range. Other available types are `TXT`, `SRV`, `MX`, `CNAME` and `A` (returning `CNAME`), in decreasing bandwidth order. Normally the “best” request type is autodetected and used. However, DNS relays may impose limits on for example `NULL` and `TXT`, making `SRV` or `MX` actually the best choice. This is not autodetected, but can be forced using the `-T` option. It is advisable to try various alternatives especially when the autodetected request type provides a downstream fragment size of less than 200 bytes.

Note that `SRV`, `MX` and `A` (returning `CNAME`) queries may/will cause additional lookups by "smart" caching nameservers to get an actual IP address, which may either slow down or fail completely.

DNS responses for non-`NULL/PRIVATE` queries can be encoded with the same set of codecs as upstream data. This is normally also autodetected, but no fully exhaustive tests are done, so some problems may not be noticed when selecting more advanced

codecs. In that case, you'll see failures/corruption in the fragment size autoprobe. In particular, several DNS relays have been found that change replies returning hostnames (`SRV` , `MX` , `CNAME` , `A`) to lowercase only when that hostname exceeds ca. 180 characters. In these and similar cases, use the `-o` option to try other downstream codecs; Base32 should always work.

Normal operation now is for the server to *not* answer a DNS request until the next DNS request has come in, a.k.a. being “lazy”. This way, the server will always have a DNS request handy when new downstream data has to be sent. This greatly improves (interactive) performance and latency, and allows to slow down the quiescent ping requests to 4 second intervals by default, and possibly much slower. In fact, the main purpose of the pings now is to force a reply to the previous ping, and prevent DNS server timeouts (usually at least 5-10 seconds per RFC1035). Some DNS servers are more impatient and will give SERVFAIL errors (timeouts) in periods without tunneled data traffic. All data should still get through in these cases, but `iodine` will reduce the ping interval to 1 second anyway (`-l1`) to reduce the number of error messages. This may not help for very impatient DNS relays like `dnsadvantage.com` (ultradns), which time out in 1 second or even less. Yet data will still get trough, and you can ignore the `SERVFAIL` errors.


If you are running on a local network without any DNS server in-between, try `-I 50` (iodine and iodined close the connection after 60 seconds of silence). The only time you'll notice a slowdown, is when DNS reply packets go missing; the `iodined` server then has to wait for a new ping to re-send the data. You can speed this up by generating some upstream traffic (keypress, ping). If this happens often, check your network for bottlenecks and/or run with `-I1` .

The delayed answering in lazy mode will cause some “carrier grade” commercial DNS relays to repeatedly re-send the same DNS query to the iodined server. If the DNS relay is actually implemented as a pool of parallel servers, duplicate requests may even arrive from multiple sources. This effect will only be visible in the network traffic at the `iodined` server, and will not affect the client's connection. Iodined will notice these duplicates, and send the same answer (when its time has come) to both the original query and the latest duplicate. After that, the full answer is cached for a short while. Delayed duplicates that arrive at the server even later, get a reply that the iodine client will ignore (if it ever arrives there).

If you have problems, try inspecting the traffic with network monitoring tools like tcpdump or ethereal/wireshark, and make sure that the relaying DNS server has not cached the response. A cached error message could mean that you started the client before the server. The `-D` (and `-DD`) option on the server can also show received and sent queries.

TIPS & TRICKS

If your port 53 is taken on a specific interface by an application that does not use it, use `-p` on iodined to specify an alternate port (like `-p 5353`) and use for instance iptables (on Linux) to forward the traffic:

```
iptables -t nat -A PREROUTING -i eth0 -p udp --dport 53 -j DNAT --to 
```

(Sent in by Tom Schouten)

Iodined will reject data from clients that have not been active (data/pings) for more than 60 seconds. Similarly, iodine will exit when no downstream data has been received for 60 seconds. In case of a long network outage or similar, just restart iodine (re-login), possibly multiple times until you get your old IP address back. Once that's done, just wait a while, and you'll eventually see the tunneled TCP traffic continue to flow from where it left off before the outage.

With the introduction of the downstream packet queue in the server, its memory usage has increased with several megabytes in the default configuration. For use in low-memory environments (e.g. running on your DSL router), you can decrease USERS and

undefine OUTPACKETQ_LEN in user.h without any ill consequence, assuming at most one client will be connected at any time. A small DNSCACHE_LEN is still advised, preferably 2 or higher, however you can also undefine it to save a few more kilobytes.

One iodine server can handle multiple domains. Set up different NS records on the same domain all pointing to the same host, and use a wildcard at the start of the topdomain argument (example *.mydomain.com). iodine will accept tunnel traffic for all domains matching that pattern. The wildcard has to be at the start of the topdomain argument and be followed by a dot.

PERFORMANCE

This section tabulates some performance measurements. To view properly, use a fixed-width font like Courier.

Measurements were done in protocol 00000502 in lazy mode; upstream encoding always Base128; iodine -M255 ; iodined -m1130 . Network conditions were not extremely favorable; results are not benchmarks but a realistic indication of real-world performance that can be expected in similar situations.

Upstream/downstream throughput was measured by scp 'ing a file previously read from /dev/urandom (i.e. incompressible), and measuring size with ls -l ; sleep 30 ; ls -l on a separate non-tunneled connection. Given the large scp block size of 16 kB, this gives a resolution of 4.3 kbit/s, which explains why some values are exactly equal. Ping round-trip times measured with ping -c100 , presented are average rtt and mean deviation (indicating spread around the average), in milliseconds.

Situation 1: Laptop -> Wifi AP -> Home server -> DSL provider -> Datacenter

iodine	DNS "relay"	bind9	DNS cache	iodined		
		downstr. fragsize	upstream kbit/s	downstr. kbit/s	ping-up avg +/-mdev	ping-up avg +/-mdev

iodine -> Wifi AP :53						
	-Tnull (= -Oraw)	982	43.6	131.0	28.0	4.6 26
iodine -> Home server :53						
	-Tnull (= -Oraw)	1174	48.0	305.8	26.6	5.0 26
iodine -> DSL provider :53						
	-Tnull (= -Oraw)	1174	56.7	367.0	20.6	3.1 21
	-Ttxt -Obase32	730	56.7	174.7*		
	-Ttxt -Obase64	874	56.7	174.7		
	-Ttxt -Obase128	1018	56.7	174.7		
	-Ttxt -Oraw	1162	56.7	358.2		
	-Tsrv -Obase128	910	56.7	174.7		
	-Tcname -Obase32	151	56.7	43.6		
	-Tcname -Obase128	212	56.7	52.4		
iodine -> DSL provider :53						
	wired (no Wifi) -Tnull	1174	74.2	585.4	20.2	5.6 19
[174.7* : these all have 2frag/packet]						

Situation 2: Laptop -> Wifi+vpn / wired -> Home server

iodine	iodined				
	downstr.	upstream	downstr.	ping-up	p:
	fragsize	kbit/s	kbit/s	avg +/-mdev	avg +/-mdev

wifi + openvpn	-Tnull	1186	166.0	1022.3	6.3	1.3	6
wired	-Tnull	1186	677.2	2464.1	1.3	0.2	1

Notes

Performance is strongly coupled to low ping times, as iodine requires confirmation for every data fragment before moving on to the next. Allowing multiple fragments in-flight like TCP could possibly increase performance, but it would likely cause serious overload for the intermediary DNS servers. The current protocol scales performance with DNS responsivity, since the DNS servers are on average handling at most one DNS request per client.

PORTABILITY

iodine has been tested on Linux (arm, ia64, x86, AMD64 and SPARC64), FreeBSD (ia64, x86), OpenBSD (x86), NetBSD (x86), MacOS X (ppc and x86, with <http://tuntaposx.sourceforge.net/>). and Windows (with OpenVPN TAP32 driver, see win32 readme file). It should be easy to port to other unix-like systems that have TUN/TAP tunneling support. Let us know if you get it to run on other platforms.

THE NAME

The name iodine was chosen since it starts with IOD (IP Over DNS) and since iodine has atomic number 53, which happens to be the DNS port number.

THANKS

- To kuxien for FreeBSD and OS X testing
- To poplix for code audit

AUTHORS & LICENSE

Copyright (c) 2006-2014 Erik Ekman yarrick@kryo.se, 2006-2009 Bjorn Andersson flex@kryo.se. Also major contributions by Anne Bezemer.

Permission to use, copy, modify, and/or distribute this software for any purpose with or without fee is hereby granted, provided that the above copyright notice and this permission notice appear in all copies.

THE SOFTWARE IS PROVIDED "AS IS" AND THE AUTHOR DISCLAIMS ALL WARRANTIES WITH REGARD TO THIS SOFTWARE INCLUDING ALL IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS. IN NO EVENT SHALL THE AUTHOR BE LIABLE FOR ANY SPECIAL, DIRECT, INDIRECT, OR CONSEQUENTIAL DAMAGES OR ANY DAMAGES WHATSOEVER RESULTING FROM LOSS OF USE, DATA OR PROFITS, WHETHER IN AN ACTION OF CONTRACT, NEGLIGENCE OR OTHER TORTIOUS ACTION, ARISING OUT OF OR IN CONNECTION WITH THE USE OR PERFORMANCE OF THIS SOFTWARE.