

# D-Link DNS-320 NAS Cr1ptT0r Ransomware ARM Dynamic Analysis - QEMU and Raspberry PI VM

Archived: 2026-04-05 21:28:03 UTC

Hi Everybody,

a few days ago I saw a tweet from @Amigo\_A\_ asking for help about a new ransomware which was affecting a D-Link 320 NAS.

The first thought was directed to the historical disabling of dlink to make sufficiently secure firmware and their willingness not to support updates. Those facts made me to think about an attack conducted over the net targetting all the devices exposed on internet itself.



Apparently was the right hypothesis.

All the users with D-Link 320XX are nowadays are at very high risk.

**TURN OFF THE DEVICE AND DISCONNECT IT FROM WAN.** On [BleepingComputer's forum](#) I asked to the affected users to check their own firmwares and trying to grab the malware. Someone did and shared the ELF on VirusTotal.

Thanks to Michael Gillespie @demonslay335 I was able to have a copy of that sample.

Hash: 9a1de00dbc07271a27cb4806937802007ae5a59433ca858d52678930253f42c1

(very few) years ago I had experience on some router exploiting and reversing (Italian ISP company named Telecom Italia and their ADSL routers), they were based on MIPS with a very good OS (Jungo OpenRG) always trivial to exploit. But this is another story, I've spent a lot of time on those devices learning some useful stuffs which today apparently become a good knowlege.

Since the fact that this ransomware is stripped (with removed debugging informations!) and statically compiled, the static analysis is very hard to do since the fact that any calls appear to be just a sub\_XXXXX because of the stripped ELF.

Because of this, we have few options to make our life less complicated:

- 1) do a dynamic analysis
- 2) create IDA pro FLIRT signatures

Starting from the first point we faced a new problem: where to run the ARM malware?

2 opportunities: the first on a QEMU VM, the second on a D-Link device (of course lol).

I do not buy D-Link stuffs, so I had only one opportunity: QEMU.



in addition, the strings shows up that he also use crypto routines from Libsodium library  
<https://libsodium.gitbook.io/doc/>

**Usage**

```
#include <libsodium.h>

int main(void)
{
    if (sodium_init() != 0) {
        return 1;
    }
    ...
}
```

libsodium.h is the only header that has to be included.

The library is called libsodium. Just -Llibsodium to linker, or obtained using pkg-config, on systems where it is available:

```
gcc -c prog.c -I$(pkg-config --cflags libsodium)
gcc -o prog prog.o $(pkg-config --libs libsodium)
```

For static linking, Visual Studio users should define \_SODIUM\_STATIC as required on other platforms.

Projects using CMake can include the Libsodium cmake

```
#include <libsodium.h>
...
crypto_secretstream_XORStream_decrypt_init(
    &ctx,
    const crypto_secretstream_XORStream_KEY *key,
    const crypto_secretstream_XORStream_AD *ad,
    const crypto_secretstream_XORStream_IV *iv);
crypto_secretstream_XORStream_decrypt(
    crypto_secretstream_XORStream_STATE *ctx,
    const crypto_secretstream_XORStream_IV *iv,
    const crypto_secretstream_XORStream_AD *ad,
    const crypto_secretstream_XORStream_CHUNK *chunk,
    crypto_secretstream_XORStream_CHUNK *out);
```

Address	Length	Type	String
00401000	0000000C	C	._systemfile
00401000	00000005	C	._rtlib
00401000	00000012	C	File is encrypted
00401000	00000016	C	File is not encrypted
00401000	00000017	C	libsodium_secretstream_decrypt
00401000	00000008	C	pubkey
00401000	00000007	C	pubkey
00401000	00000022	C	encrypting using public key: %s\n
00401000	00000023	C	decrypting using private key: %s\n
00401000	00000007	C	done\n
00401000	00000009	C	Yes, Prng
00401000	00000009	C	Yes, Prng
00401000	00000016	C	_of_supported
00401000	0000001C	C	._PEBS_ENGINES_SUPPORT_LIBNAME.txt
00401000	00000016	C	._of_supported
00401000	00000013	C	._of_supported
00401000	00000004	C	pubkey
00401000	00000009	C	my_hexbin
00401000	00000006	C	._JvlibC
00401000	00000004	C	hex
00401000	00000022	C	error: badhex: malloc: could not allocate memory\n
00401000	00000011	C	error: badhex: %s\n
00401000	0000000C	C	[%s] [%s]\n
00401000	00000018	C	broken_commit == broken
00401000	00000018	C	invalid hex value: %s\n
00401000	00000019	C	invalid sized size: %s\n
00401000	00000009	C	seed
00401000	00000023	C	error opening %s for reading: %s\n
00401000	00000023	C	error opening %s for writing: %s\n
00401000	00000014	C	error opening file
00401000	00000009	C	%s\n
00401000	00000008	C	._of_supported
00401000	00000007	C	%s.tmp
00401000	00000010	C	encrypted: %s\n
00401000	00000008	C	file corrupt
00401000	00000038	C	message corrupted or not intended for this recipient: %s\n
00401000	00000004	C	%s output
00401000	00000014	C	incomplete header\n
00401000	00000012	C	corrupted chunk\n
00401000	00000044	C	premature end (end of file reached before the end of the stream)\n
00401000	00000010	C	decrypted: %s\n
00401000	0000000C	C	..decrypt.c
00401000	00000009	C	ret. = 0
00401000	00000018	C	0x401000 (hex) 0x00000000 (dec)

Like I said before, the - hardest - next step is to create a IDA FLIRT signature, by cross compiling some example from Libsodium repo (hoping that it will use the same functions as the malware), extract the signatures by using FireEye idb2pat tool [https://www.fireeye.com/blog/threat-research/2015/01/flare\\_ida\\_pro\\_script.html](https://www.fireeye.com/blog/threat-research/2015/01/flare_ida_pro_script.html) to have an understandable static analysis to MAYBE retrieve the private key and decrypt the files, or at least have a reduced key space to make possible a brute force attack.

Follow me on Twitter and I'll keep you updated.

Cheers

RE Solver

Source: <https://resolverblog.blogspot.com/2019/02/d-link-dns-320-nas-cr1ptt0r-ransomware.html>