From Zero To 50k Infections - PseudoManuscrypt Sinkholing - Part 1

By Stanislas Arnoud

Published: 2022-10-05 · Archived: 2026-04-05 20:30:50 UTC

At Bitsight we do a lot of malware sinkholing, and by late 2021 we started registering some DGA-like domains that not only did not belong to any known domain generation algorithm (DGA) but were also being classified as different types of malware like SmokeLoader, PrivateLoader, Socelars, and Redline. We registered more than 50 domains, all in the form [a-jm-r]{10}.com, and started investigating what we thought could be an unknown malware family.

In this post we'll go through a technical analysis of this unknown malware, describing how we went from unknown DGA-like domains to sinkholing and emulating a fairly recent botnet that in the last 8 months has infected nearly 500,000 machines (2.2M unique IPs) in total, across at least 40 countries, and has a current estimated botnet size of around 50,000 machines.

The process of tracing back a domain to a specific malware sample that generates traffic can vary; It can sometimes be as easy as opening the first result on a Google search for the domain, or as annoying as going through multiple search engines and ending on a Wayback Machine page that does not resolve. In this section we'll detail how we traced a sample for this specific malware family, going from the domains and their traffic to a recent family dubbed PseudoManuscrypt by Kaspersky.

We were confident that we had registered DGA domains given the pattern and traffic of the domains. We also knew that the domains were generating UDP traffic on port 53 (Figure 1), so we started by looking at communicating files from VirusTotal Intelligence (Figure 2) to get a better sense of the infection chain and hopefully reach a sample that would communicate with our sinkholes.

Start Time	Stop Time	Src IP / Country	Src Port	Dst IP / Country	Dst Port	Packets	Databytes / Bytes
udp			39981		53	9	100 352
udp			62218		53	8	96 320
udp			46225		53	12	124 460
udp			58358		53	8	96 320
udp			52894		53	12	124 460
udp			48293		53	8	96 320
udp			56421		53	12	124 460
udp			22412		53	8	96 320
udp			60148		53	9	108 360
udp			63779		53	8	96 320
udp			48819		53	12	124 460

Figure 1

Our first impression was that there were a lot of files generating traffic to the DGA (Figure 2) and that many were installers or archives that contained detections for multiple families, explaining why our systems' classification was also reporting different families for the set of domains.

URLs (2)

Scanned	Detections	Status	URL
2022-05-19	6 / 92	200	http://jggrmmojcc.com/
2021-12-14	6 / 93	-	https://jggrmmojcc.com/

Communicating Files (346)

Scanned	Detections	Type	Name
2022-07-03	52 / 68	Win32 EXE	setup_x86_x64_install.exe
2022-05-13	48 / 66	Win32 EXE	setup_x86_x64_install.exe
2021-10-20	48 / 67	Win32 EXE	70870cf28b7e34965164f88d013f1427.virus
2022-04-21	47 / 67	Win32 EXE	C:\Users\user\AppData\Local\Temp\RarSFX0\shulanli.exe
2021-11-28	52 / 68	Win32 EXE	7zS.sfx
2022-08-24	60 / 71	Win32 EXE	TrySearch.exe
2021-10-20	45 / 68	Win32 EXE	setup_x86_x64_install.exe
2021-10-11	46 / 66	Win32 EXE	setup_x86_x64_install.exe
2021-10-20	38 / 67	Win32 EXE	%TEMP%\setup_installer.exe
2022-06-03	49 / 68	Win32 EXE	7zS.sfx

Figure 2.

The large amount of bundled communicating files makes it harder to track down the exact source of the traffic, so we went through some trial and error, sending multiple files to a sandbox and searching both for traffic to the DGA domains, as well as the uncommon UDP traffic on port 53. While going through the sandbox's runs we found this sample, which contains UDP traffic on port 53 to toa.mygametoa[.]com. By searching for this domain on Google, we found a fairly recent (by the time of the research) article from Kaspersky mentioning a new malware family: PseudoManuscrypt.

The article's description of how PseudoManuscrypt is distributed and communicates with the C2 matches what we were seeing, and it filled the gap for some of our unknowns: the uncommon UDP traffic we were seeing was actually KCP over UDP, and the kill chain that led to the communication was originating from a .dat and .dll file. Looking back at this sample, we could see a call to rundll32.exe with a file ending in sqlite.dll. This file (together with the .dat) was being downloaded by one of the bundled executables from t.gogamec[.]com (Figure 3).

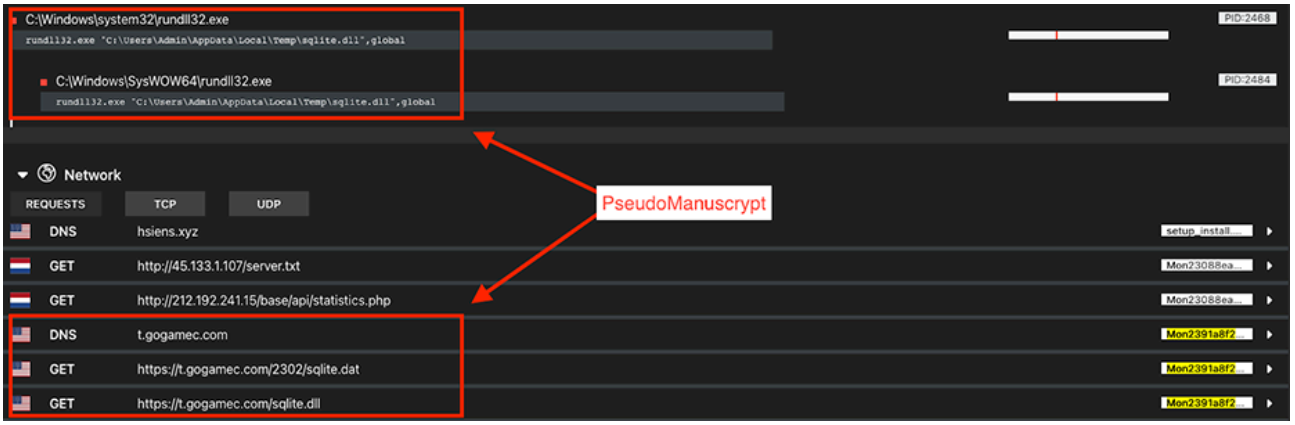

Figure 3.

The article gave us high confidence that we were seeing PseudoManuscrypt traffic to our sinkholes. There was, however, no mention of any Domain Generation Algorithm within the research.

Distribution and Execution Flow

In the previous section we found the malware family that was generating the traffic we were observing, and we also identified one of the distribution methods. In this section we'll dig further into the distribution and execution flow of PseudoManuscrypt, enabling us to easily obtain recent samples of it, as well as giving us a starting point for a more in-depth analysis of the communication protocol.

We already know based on Kaspersky's work that there are multiple distribution methods for the PseudoManuscrypt family. For us it was obvious from the beginning that some of it is done from within archive files containing many other malware families, like Socelars, Smokeloader or Redline (hence our DGA domains being incorrectly classified), but during some parallel work we had on PrivateLoader, we noticed samples that included a hardcoded URL to fetch the main PseudoManuscrypt executable (Figure 4).

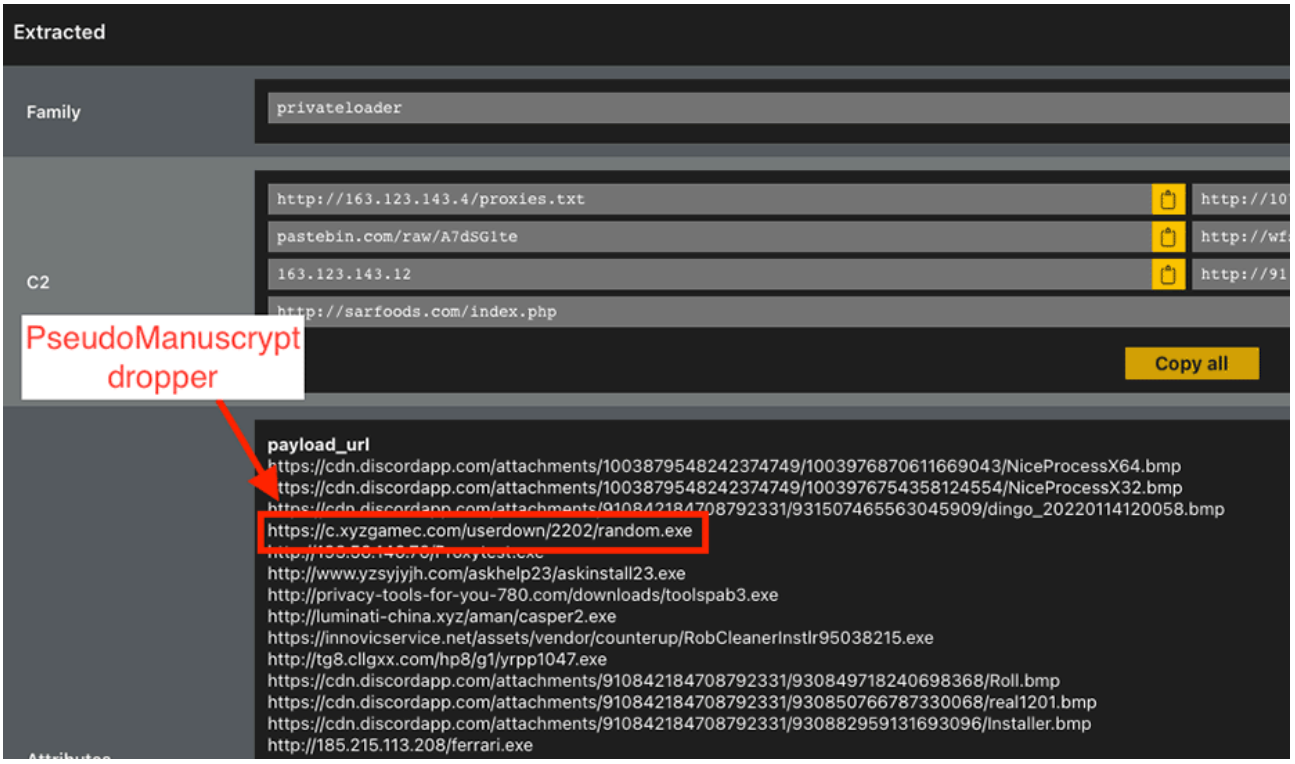

Figure 4.

We do not know the purpose of the hardcoded URL list that PrivateLoader has, since we didn't see PrivateLoader samples fetching those URLs, but the presence of this URL hints at the possibility of PrivateLoader distributing PseudoManuscrypt. At the time of writing the URL redirected to another domain (b.dxyzgame[.]com) that wasn't resolving to any IP, but in the past it was dropping this executable.

Both the previously mentioned executable and the executables bundled in the installers and archives show signs of being a dropper for the PseudoManuscrypt family. Their behavior coincides with Kaspersky's description as well as with what we've seen, where the executable downloads 2 files from a domain that, as far as we've seen, always contains the "game" keyword (check IOCs for domain list).

As for the files that are downloaded:

- One is a packed data file, usually pointed by the URI /X/sqlite.dat or X.html where we believe that X is the campaign number.
- The other file is a 32-bit Windows library, pointed by the URI /sqlite.dll or login.html.

Most recent droppers write both files to %APPDATA%\Local\Temp with the names sqlite.dll and sqlite.dat, and then run the following command:

```
rundll32.exe "C:\Users\X\AppData\Local\Temp\sqlite.dll",global
```

The "global" export will locate the file named sqlite.dat in the same directory, unpack it in memory and then execute it (Figure 5). The unpacking algorithm includes a rolling XOR, and a decompression routine. sqlite.dll runs the rolling XOR on sqlite.dat, and then jumps to the beginning of it. The decrypted shellcode will then XOR

and decompress a part of itself using the LZNT1 algorithm. We've shared a Python script to unpack the .dat file here.

```
1
2  void __fastcall fn_xor_with_byte(byte *array,uint size)
3
4  {
5    uint uVar1;
6
7    if ((1 < size) && (uVar1 = 0, size != 0)) {
8      do {
9        if ((uVar1 & 1) == 0) {
10         array[uVar1] = array[uVar1] ^ 0xa7;
11       }
12       else {
13         array[uVar1] = array[uVar1] ^ 0x6a;
14       }
15       uVar1 = uVar1 + 1;
16     } while (uVar1 < size);
17   }
18   return;
19 }
```

```
1
2  void __fastcall fn_xor_with_prev(byte *array,uint size)
3
4  {
5    int iVar1;
6
7    if (1 < size) {
8      array[size - 1] = array[size - 1] ^ *array;
9      for (iVar1 = size - 2; iVar1 != 0; iVar1 = iVar1 + -1) {
10       array[iVar1] = array[iVar1] ^ array[iVar1 + 1];
11     }
12     *array = *array ^ array[1];
13   }
14   return;
15 }
```

```
34  DAT_1000b930 = fn_wrapper_HeapCreate();
35  if ((_is_kernel32_loaded & 1) == 0) {
36    _is_kernel32_loaded = _is_kernel32_loaded | 1;
37    handle_kernel32 = LoadLibraryA("KERNEL32.DLL");
38  }
39  ptr_HeapAlloc = GetProcAddress(handle_kernel32,&str_HeapAlloc);
40  if (ptr_HeapAlloc != (FARPROC)0x0) {
41    fn_code = (code *)(*ptr_HeapAlloc)(DAT_1000b930,0,0x100000);
42    outsize = 0;
43    fn_decrypt_file_inmem(fn_code,&outsize,filename);
44    if (fn_code != (code *)0x0) {
45      (*fn_code)();
46    }
47  }
48  return 0;
```

Figure 5.

This execution flow would suggest that the DLL that is downloaded simply behaves as the unpacker for the packed core component. This claim is backed by the fact the unpacker DLL is always identical, whereas the packed core changes frequently.

We close this section with a better understanding of the distribution and execution flow of this family (Figure 6), and with easy access to the core component that will enable us to better understand the communication protocol.

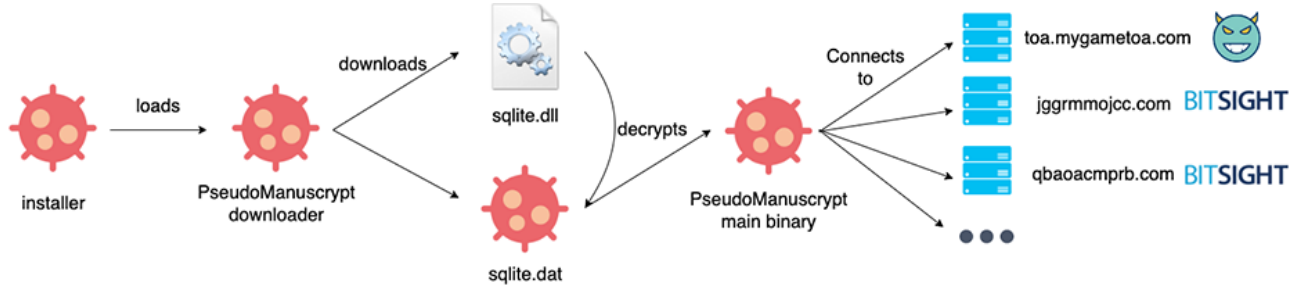

Figure 6.

In this section we'll detail some of the analysis we did on the core component of PseudoManuscrypt. We won't detail every aspect of the sample, as our main focus was on understanding the basics of the communication protocol to create a custom sinkhole.

Hardcoded C2s and DGAs

By this point we had access to PseudoManuscrypt's core component, and our hypothesis was that there was a main (possibly hardcoded) C2 domain (toa.mygametoa[.]com) and a fallback DGA that would trigger on specific conditions. To quickly test our hypothesis, we manually ran the unpacker and the core, while also changing the hosts file to point toa.mygametoa[.]com to 127.0.0.1. This triggered the DGA (Figure 7), confirming our hypothesis.

```
16977 231.056206 [redacted]  DNS  74 Standard query 0x7072 AAAA jggrmmojcc.com
16980 231.204563             DNS  90 Standard query response 0x3c11 A jggrmmojcc.com A [redacted]
16981 231.206089             DNS  74 Standard query response 0x7072 AAAA jggrmmojcc.com
```

Figure 7.

We then started looking at the core component itself to find and reverse the DGA. Just by looking at the strings contained in the core, we were able to identify two hardcoded C2s:

- toa.mygametoa[.]com
- tob.mygametob[.]com

By looking for references to the hardcoded C2s, we identified that the client tries to connect to the first one, if it is not able to communicate with it, the infected machine will fallback to the DGA. As for the second hardcoded domain, our understanding is that it will be used only if the C2 server instructs the client to use it.

The DGA uses the hardcoded C2 and a key as seeds. The algorithm is as follows (Figure 8):

1. The domain and key variables are concatenated with a ","
2. The string is MD5 hashed
3. 8 bytes are retrieved from the hash, starting at index 4 and ending at position 12
4. For the first 10 hexadecimal values of the retrieved bytes:
    1. If it's a number, 0x31 is added (outputs "a" to "j")
    2. If an uppercase letter, 0x2c is added (outputs "m" to "r")
5. ".com" is appended to the resulting string

For every call to the DGA algorithm, the previously generated domain is used as the first argument, while the second argument is always the same string. This makes the domains dependent on the previously generated domain, making a circular group of all possible domains.

A Python implementation of this DGA is available here.

```
wsprintfW((LPWSTR)seed,L"%s,%s",url,key);
ptr_end_seed = (int)seed;
do {
  sVar2 = *(short *)ptr_end_seed;
  ptr_end_seed = ptr_end_seed + 2;
} while (sVar2 != 0);
fn_hash_md5(seed,(ptr_end_seed - (int)(seed + 2) >> 1) * 2,(byte *)hash);
_memset(url,0,200);
if ((hash[0] < '0') || ('9' < hash[0])) {
  if (('@' < hash[0]) && (hash[0] < '[')) {
    hash[0] = hash[0] + 0x2c;
  }
}
else {
  hash[0] = hash[0] + 0x31;
}
*url = (short)hash[0];
```

Figure 8.

The infected client will try to connect to each of the sequentially generated domains, and everytime it fails it will generate a new one.

Given the circular characteristics of the DGA and the small size of the domains, we generated all possible domains for both the hardcoded C2s. Here you can see a list with all possible 585,723 unique domains for the toa.mygametoa[.]com C2, and here is a list with all possible 812,811 unique domains for the tob.mygametob[.]com C2.

In order to collect infection telemetry from infected machines, we needed to implement the protocol the malware uses.

Communication protocol

From Kaspersky's research and our own, we know that this family communicates using KCP over UDP, and TCP as a fallback. The KCP protocol has been designed as a replacement for TCP, and its specification can be found on GitHub. PseudoManuscrypt's code of KCP is very similar to the linked repository, suggesting that they might be using that specific implementation.

Although the communication protocol uses UDP port 53 and TCP port 443, they implement their own messaging protocol. We focused on the first exchange between the client and the server, which allowed us to map connections from the domains into infections, and also extract a lot of information about the infected machines.

This custom protocol can be divided into 2 layers. The first layer (L1) contains the message to be sent, while the second layer (L2) contains metadata about the message. Both layers are independent of each other.

L2 has the following format:

```
Struct L2
{
BYTE header;
BYTE compression_type;
int32 L2_size;
int32 L1_size;
BYTE L1[L2_size - 10];
}
```

Where:

- `header` byte is always 0x43
- `compression_type` can be one of the following:
    - 0x0F -> no compression, no encryption
    - 0x1F -> no compression, L1 xored with 0x88
    - 0x2F -> zlib compression, no encryption
    - 0x3F -> zlib compression, then L1 xored with 0x88
    - 0x4F -> RtlCompressBuffer compression, no encryption
    - 0x5F -> RtlCompressBuffer compression then L1 xored with 0x88

- `L2_size` is the size of L2, which contains both the metadata and the compressed L1
- `L1_size` is the real size of L1
- `L1` is an array with L1 content

The L1 format will vary based on the type of received message. We believe that the generic format is the following:

```
Struct L1
{
BYTE message_type;
... varies based on message_type
}
```

First message

When the infected machine first communicates with the C2, it sends a lot of information about the computer. In all samples we found, the `compression_type` was always 0x3F (zlib compression, then 1-byte-xor with 0x88) for the L1 format. The following is what we believe is the L1 format for this initial message:

```
struct L1_layer
{
BYTE message_type;
BYTE padding1[3];
int32 campaign;
char client_id[33];
BYTE padding2[3];
int32 major_winver;
int32 minor_winver;
int32 build_number;
int32 platform_id;
int32 is_intel_amd;
int32 is_running_wow64;
int32 winserv_version; // Not working (always 0)
int32 is_winserv;
BYTE padding3[256];
int16 number_of_proc;
BYTE padding4[2];
int32 proc_freq_mhz;
int32 mem_size_mb;
char hostname[50];
BYTE padding5[2];
int32 ms_needed_to_connect;
BYTE tbd2[100];
wchar client_release_date[9];
```

```
BYTE padding6[182];
wchar client_version[9];
BYTE padding7[22];
int32 has_a_camera;
char internal_ip_port[94];
int16 port;
int32 tickcount;
int32 pid;
int32 sz_firmware_table;
BYTE firmware_table[sz_firmware_table];
}
```

Most of the fields are self-explanatory, we'd just like to detail that:

- `message_type` is 0x99 for this first message
- `paddingX` fields are always null
- `campaign` always matches the number that is seen in the URL from there the sqlite.dat was downloaded from
- `client_id` is the MD5 sum of the substructure containing the fields `sz_firmware_table` and `firmware_table`
- `client_release_date` and `client_version` are dates and are almost always the value, with the exception that `client_version` has a `v` in the beginning.

First response

Once the server receives the first message, it can send various types of commands to execute on the system, like: running a binary, clearing event logs, modifying registry keys, etc. The C2 usually responds with a sleep command of 3600 seconds (`message_type` 0x00), unless the `client_version` field in the first message isn't the most recent one, and if so the C2 replies with a "binary update" (`message_type` 0x01) command, and sends the most recent version of PseudoManuscrypt's core.

We did not investigate the communication protocol any further, as at this point we had enough information to be able to identify and sinkhole PseudoManuscrypt infections.

With the previous work we developed our own custom sinkhole for this family. We've been tracking it for over 8 months, seeing a daily botnet size of around 16,000 machines until the end of August, where a new version with a new C2 was pushed (consequently a new DGA) and the daily numbers dropped to around 7k.

Roughly 5 days before the new version was pushed, we saw a sudden increase in infections that peaked at around 51,500 unique client IDs (Figure 9). We hypothesize that the hardcoded C2 could have had an issue, making the bots fallback to the DGA, and consequently communicate with our sinkhole.

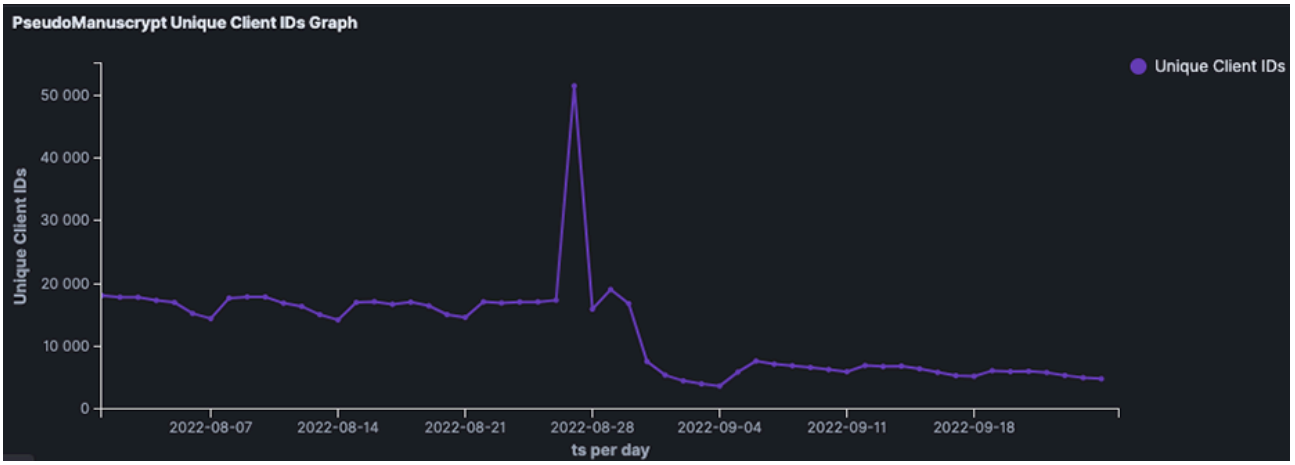

Figure 9.

As PseudoManuscrypt seems to be a recent, but fairly large botnet, in a future post we'll detail some insights about the telemetry we're getting from sinkholing the DGA domains. We'll also detail the botnet's evolution and its bots.

VT Graph
sqlite.dll dd19804b5823cf2cab3afe4a386b427d9016e2673e82e0f030e4cff74ef73ce1 sqlite.dat ecdfa028928da8df647ece7e7037bc4d492b82ff1870cc05cf982449f2c41786

toa.mygametoa[.]com
tob.mygametob[.]com
b.dxyzgame[.]com
56.jpgamehome[.]com
gp.gamebuy768[.]com
v.xyzgamev[.]com
c.xyzgamec[.]com
xv.yxzgamen[.]com
g.agametog[.]com
