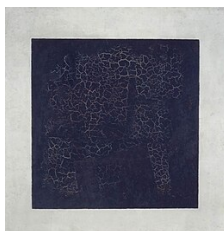


Analysis of a Low-Detection Linux Implant with Hands-On Intrusion Capabilities

 dmpdump.github.io/posts/Unattributed_Linux_Implant

July 16, 2026



UPDATE: After reviewing related infra found by [@500mk500](#) and a related sample by [@malwrhunterteam](#), I found quite a few similarities with the Adaptix Agent using the `adaptix_gopher` protocol.

In early July 2026, [MalwareHunterTeam](#) shared an interesting ELF named `gregbfdah.png` with minimal detection in VirusTotal. The ELF was uploaded to VirusTotal on 2026-06-16 and it still has very low detection. The file was delivered from the following AWS domain: `https://zapier-logos.s3.amazonaws[.]com/gregbfdah.png`. That URL was first submitted to VirusTotal from India on 2026-06-11.

The file in question is a statically-linked x86-64 Linux ELF backdoor/C2 implant written in C++, possibly still in development. When executed, the backdoor:

- Implements a classic double fork for daemonization.
- Redirects its standard streams to `/dev/null`.
- Changes directories to `/tmp`.
- Checks if it was executed with the `-node1` argument. If it was, it executes the main backdoor logic. If not, it self-deletes.

```

sub_4911AC(13LL, 1LL);
v0 = fn_fork(); // first fork
if ( v0 >= 0 )
{
    if ( v0 )
        return 0LL; // parent exits
    fn_setsid();
    if ( (int)fn_fork() > 0 ) // second fork
        sub_49FA1E(0LL); // second parent exits
    v5 = fn_open((unsigned int)"/dev/null", 2, v1, v2, v3, v4);
    v6 = v5;
    if ( v5 >= 0 )
    {
        sub_4988AA((unsigned int)v5, 0LL);
        sub_4988AA(v6, 1LL);
        sub_4988AA(v6, 2LL);
        if ( (int)v6 > 2 )
            sub_49886D(v6);
    }
    fn_chdir("/tmp");
}
fn_check_arg_delete(); // check if argument is -nodet, if no self-delete
while ( (unsigned int)fn_main_function() )
    fn_sleep(10LL);
return 0LL;

```

Configuration Decryption

The implant uses the [MessagePack](#) binary serialization format for its configuration and C2 communication. The main function of the malware starts by retrieving, decrypting, and parsing its configuration.

The following attributes are parsed from the configuration:

- type
- banner_size
- conn_timeout
- conn_count
- sleep_delay
- jitter
- addresses

```

}
fn_retrieve_encrypted_config(&v261, a2, a3); // Retrieves embedded configuration
if ( (unsigned __int64)(v262 - v261) > 0xF )
{
    fn_AES_GCM_decrypt((__int64 *)&v265, (unsigned __int64 *)(v261 + 16), v262 - v261 - 16, v261, 0x10uLL); // Decrypts embedded configuration
    v6 = (unsigned __int64)v265;
    v7 = v266;
    fn_zero_out_structure(&v269);
    if ( (unsigned int)fn_MsgPack_deserializer(&v269, v6, (__int128 *)((char *)v7 - v6)) == 2 && (_DWORD)v270 == 7 )
    {
        v8 = (unsigned int *)fn_key_lookup((__int64)&v271, (__int64)"type");
        if ( v8 )
            sub_419643(v8, &v273);
        v9 = fn_key_lookup((__int64)&v271, (__int64)"banner_size");
        if ( v9 && sub_405F43(v9, &v259) )
            v276 = v259;
        v10 = fn_key_lookup((__int64)&v271, (__int64)"conn_timeout");
        if ( v10 && sub_405F43(v10, &v259) )
            v277 = v259;
        v11 = fn_key_lookup((__int64)&v271, (__int64)"conn_count");
        if ( v11 && sub_405F43(v11, &v259) )
            v278 = v259;
        v12 = (unsigned int *)fn_key_lookup((__int64)&v271, (__int64)"sleep_delay");
        if ( v12 )
            sub_419643(v12, &v279);
        v13 = (unsigned int *)fn_key_lookup((__int64)&v271, (__int64)"jitter");
        if ( v13 )
            sub_419643(v13, (_DWORD *)&v279 + 1);
        v14 = (_DWORD *)fn_key_lookup((__int64)&v271, (__int64)"addresses");
        v15 = v14;
        if ( v14 && *v14 == 6 )
        {
            sub_406988((__int64 *)&v274, (unsigned int)v14[2]);
            for ( i = 0; i < v15[2]; ++i )
                sub_419624*((_QWORD *)v15 + 2) + 24LL * i, (__int64)&v274[4 * i]);
        }
        v5 = 1;
        sub_405D09(&v269);
    }
}

```

The configuration is embedded in the .data section and is encrypted with AES-128 in GCM mode. The blob with the encrypted configuration has the following structure:

- The first 16 bytes are the key.
- The next 12 bytes are the nonce.
- Bytes 28 until the tag is the encrypted configuration.
- The last 16 bytes are the tag.

	nonce	key
00000000:	A0 5E BE 9E-D6 04 23 A6-AB EC 5B 03-FD 47 DC 86	á^J R_♦#≡%∞[♥²G_â
00000010:	D1 BA C7 7F-6C C9 E1 80-80 79 3E 93-6E 28 7B 59	τ Δl fßÇÇy>ðn({Y
00000020:	F3 99 90 ED-C2 9A 34 CA-02 28 96 A0-57 CF 5E 7F	≤ÖÉçÜ4=0(ûâw=^Δ
00000030:	1B DB 30 5B-5F 2B 2A BD-70 BC 4B 9B-C6 75 E5 C4	+■0[_+* p Kç uσ-
00000040:	A5 37 8D 2C-E8 1E 39 17-EB 6C C2 52-9D C6 F2 56	Ñ7î,0▲9±δlτR¥ ≥V
00000050:	58 8C 6B E2-09 2E 6E 3F-36 AF 77 90-72 DF BA 98	XîkΓo.n?6»wÉr■ ÿ
00000060:	53 15 82 BA-73 50 F3 FC-6F A0 51 04-C2 A6 45 FA	SŞé sP≤°oáQ♦τ≡E·
00000070:	BC 2A 3A F4-CC 00 54 18-47 1B E1 2C-80 AE 0D 97	*: f T†G+ß,Ç« ù
00000080:	DC 33 6F 81-E2 98 CC 9E-87 78 8D 79-8B 4B 53 3D	3ouŕŷ HçxiyiKS=
00000090:	FC B9 C3 19-06 AC - - - - - - - -	n= ~J.¼

The following script can be used to decrypt and parse the configuration once the embedded blob has been extracted:

```

from cryptography.hazmat.primitives.ciphers.aead import
AESGCM
import msgpack

with open("extracted_blob.bin", "rb") as f:
    raw = f.read()

blob = raw[:150]
key = blob[:16]
nonce = blob[16:28]
encrypted_config = blob[28:-16]
tag = blob[-16:]

print(f"Key:    {key.hex()}")
print(f"Nonce: {nonce.hex()}")
print(f"CT:     {encrypted_config.hex()}")
print(f"Tag:    {tag.hex()}")

aesgcm = AESGCM(key)
plaintext = aesgcm.decrypt(nonce, encrypted_config + tag,
None)

print(f"\n[+] Decrypted {len(plaintext)} bytes")
print(f"Raw hex: {plaintext.hex()}")

config = msgpack.unpackb(plaintext, raw=False)
print("\n[+] C2 Config:")
for k, v in config.items():
    print(f"  {k}: {v}")

```

Decrypted and parsed configuration for this backdoor:

```

Key: a05ebe9ed60423a6abec5b03fd47dc86
Nonce: d1bac77f6cc9e18080793e93
CT: 6e287b59f39990edc29a34ca022896a057cf5e7f1bdb305b5f2b2abd70bc4b9bc675e5c4a5378d2ce81e3917eb6cc2529dc6f2
56588c6be2092e6e3f36af779072dfba98531582ba7350f3fc6fa05104c2a645fabcc2a3af4cc005418471be12c80ae0d97dc336f81e29
8
Tag: cc9e87788d798b4b533dfcb9c31906ac

[+] Decrypted 106 bytes
Raw hex: 87a474797065ce904e5493a961646472657373657391b3696f742e3938313636362e78797a3a38303830ab62616e6e65725f
73697a6507ac636f6e6e5f74696d656f75740aaa636f6e6e5f636f756e74ce3b9aca00ab736c6565705f64656c617900a66a697474657
200

[+] C2 Config:
  type: 2421052563
  addresses: ['iot.981666.xyz:8080']
  banner_size: 7
  conn_timeout: 10
  conn_count: 1000000000
  sleep_delay: 0
  jitter: 0

```

Next, the agent generates a per-host ID (a “fingerprint”) using SHA1. The fingerprint is comprised of the following elements:

- The nginx string. This does not change.
- The machine id from /etc/machine-id. If that fails, it tries with /proc/sys/kernel/random/boot_id.
- The username (via the \$USER and \$LOGNAME environment variables).
- The MAC address, enumerating interfaces under /sys/class/net, searching for the first non-loopback interface.
- The reverse static string. This does not change.

```

{
    v17 = 1;
    goto LABEL_312;
}
v311[0] = 0xEFCDA88967452301LL; // initialize SHA1 state
v311[1] = 0x1032547698BADCFELL;
v312 = -1009589776;
v313 = 0LL; // Agent ID generation (SHA1)
fn_SHA1_hash_update((__int64)v311, "nginx", 5uLL); // "nginx" used as seed
sub_412E1A(v309, (__int64)"/etc/machine-id"); // reads /etc/machine-id
if ( v309[1] )
    sub_44F0B0(v280, v309);
else
    sub_412E1A(v280, (__int64)"/proc/sys/kernel/random/boot_id"); // fallback is /proc/sys/kernel/random/boot_id
fn_sys_munmap(v309);
fn_SHA1_hash_update((__int64)v311, v280[0], (unsigned __int64)v280[1]);
v18 = (const char *)sub_48A4D4("USER");
v19 = v18;
if ( !v18 || !*v18 )
{
    v20 = (const char *)sub_48A4D4("LOGNAME");
    v19 = v20;
    if ( !v20 || !*v20 )
    {
        v21 = sub_4989DA("LOGNAME", v20);
        v22 = (const char **)sub_49092B(v21);
        if ( !v22 || (v19 = *v22) == 0LL )
            v19 = &filename;
    }
}
std_str(v281, (__int64)v19);
fn_SHA1_hash_update((__int64)v311, v281[0], (unsigned __int64)v281[1]);

```

```

v46 = v270;
for ( k = v269; v46 != (__int64 *)k; k += 32LL )
{
    std_str(&v285, (__int64)"/sys/class/net");
    sub_4082BB((__int64)&v287, (__int64)&v285, (__int64)"/");
    sub_4082DA((__int64)&v289, (__int64)&v287, k);
    sub_4082BB((__int64)v309, (__int64)&v289, (__int64)"/");
    sub_4082BB((__int64)&v282, (__int64)v309, (__int64)"address");
    fn_sys_munmap(v309);
    fn_sys_munmap((__int64 *)&v289);
    fn_sys_munmap((__int64 *)&v287);
    fn_sys_munmap((__int64 *)&v285);
    sub_412E1A(v309, (__int64)v282);
    if ( v309[1] && (v309[1] != 17 || (unsigned int)sub_494E40((__BYTE *)v309[0], "00:00:00:00:00:00", 17LL)) )
    {
        sub_44F0B0(&v289, v309);
        fn_sys_munmap(v309);
        fn_sys_munmap((__int64 *)&v282);
        goto LABEL_82;
    }
    fn_sys_munmap(v309);
    fn_sys_munmap((__int64 *)&v282);
}
std_str(&v289, (__int64)&filename);
LABEL_82:
sub_404B74(&v269);
}
else
{
    std_str(&v289, (__int64)&filename);
}
fn_SHA1_hash_update((__int64)v311, v289, v290);
fn_SHA1_hash_update((__int64)v311, "reverse", 7uLL);
LOBYTE(v265) = 0x80;
v47 = 8 * v313;
fn_SHA1_hash_update((__int64)v311, (char *)&v265, 1uLL);
LOBYTE(v269) = 0;
while ( (v313 & 0x3F) != 0x38 )
    fn_SHA1_hash_update((__int64)v311, (char *)&v269, 1uLL);

```

Only the first 4 bytes of the SHA1 digest are extracted as the agent ID, with `_byteswap_ulong` applied for endian conversion.

Key Generation and System Profiling

Next, the 'agent' generates a new random AES-128 session key and retrieves the key from the configuration (for further use).

```

fn_random_generator(&session_key, 16LL); // create random session key
fn_retrieve_encrypted_config(&v286, 16LL, v59);
v61 = v286;
if ( (unsigned __int64)&v287[-v286] > 0xF )
{
    key_from_config = 0LL;
    v246 = 0LL;
    v247 = 0LL;
    sub_419A08(&key_from_config, (_BYTE *)v286, v286 + 16, v60); // extracts config key
}
else
{
    v61 = 16LL;
    LOBYTE(v282) = 0;
    sub_411ED6(&key_from_config, 16LL, (char *)&v282);
}
sub_40479C(&v286);
sub_406BA4((u32)&unk_4CC480, v61, v62);
qword_4CC478 = (__int64)&v290;
sub_419B3C(&qword_4CC460, (__int64 *)&key_from_config);
sub_412DAD((__int64)&unk_4CC480);

```

After generating the session key, the agent starts profiling the infected host. The following items are retrieved:

- The process name that the agent is running as by checking /proc/self/exe. If it cannot be retrieved, it falls back to agent.
- The PID (process ID).
- The username by first getting the user ID and looking it up in /etc/passwd. Fallbacks include the \$USER environment variable, the \$LOGNAME environment variable, or unknown if everything else fails.
- The hostname by retrieving the nodename via the uname syscall.
- The IP address is retrieved by creating a UDP socket (using 8.8.8.8 as a dummy address), calling connect to trigger a routing table lookup, and retrieving the local IP address via getsockname. It falls back to 127.0.0.1.
- Validates if it is running as root by comparing the uid to 0.
- The OS type is always set to "linux".
- OS information by parsing /etc/os-release and parsing NAME=, VERSION=, and VERSION_ID=. This check falls back to running uname, or "Linux" if everything else fails.

Process name retrieval and fallback to agent:

```

v57 = sub_498A3A("/proc/self/exe", (char *)v311, 4095LL); // Gets its own process name
if ( v57 <= 0 )
{
    v59 = (__int64 *)"agent"; // fallback name: "agent"
}

```

Username retrieval:

```

pid = fn_get_pid(); // Gets PID
v60 = fn_geteuid(); // Gets EUID
v61 = (const char **)fn_lookup_euid_etc_passwd(v60); // looks up uid in /etc/passwd
if ( v61 )
{
    v62 = *v61;
}
else
{
    v62 = (const char *)sub_48A4D4("USER"); // fallback: environment var $USER
    if ( !v62 )
    {
        v62 = (const char *)sub_48A4D4("LOGNAME"); // fallback: environment var $LOGNAME
        if ( !v62 )
            v62 = "unknown";
    }
}
std_str(v311, (__int64)v62);
sub_419970(v293, v311);
fn_sys_munmap(v311);
v63 = v311;
for ( m = 64LL; m; --m )
{
    *(_DWORD *)v63 = 0;
    v63 = (__int64 *)((char *)v63 + 4);
}
v65 = (__int64 *)"unknown"; // fallback: "unknown"

```

Hostname retrieval:

```

{
    __int64 result; // rax
    __int64 v4; // rsi
    __int64 i; // rdx
    char v6; // cl
    struct old_utsname v7; // [rsp+0h] [rbp-1A8h] BYREF

    result = fn_sys_uname(&v7);
    if ( (_DWORD)result )
        return 0xFFFFFFFFLL;
    v4 = 65LL;
    if ( a2 <= 0x41 )
        v4 = a2;
    for ( i = 0LL; i != v4; ++i )
    {
        v6 = v7.nodename[i];
        *(_BYTE *)(a1 + i) = v6;
        if ( !v6 )
            return result;
    }
    if ( i )
        *(_BYTE *)(a1 + i - 1) = 0;
    return result;
}

```

IP retrieval:



OS type and OS information:



Beacon Registration

After profiling the system, the agent builds a beacon to register to the C2. The beacon includes a combination of configuration settings and system profiling information. The registration beacon is serialized using the MessagePack format. The registration MessagePack has an outer map and an inner map.

The inner map contains: the process name, the pid, the user, the hostname, the ip address, whether or not the implant is running as root, the os, the os version, the encryption key (the previously generated session key), the sleep delay, and the jitter. In addition to these values, which were observed before in the system profiling and configuration, additional values are sent:

- acp
- oem
- rootkit_scheme

All three are set to 0, and not modified. While I did not see a clear explanation for these fields in the implant, a hypothesis is that there may be versions of the agent that support Windows. acp could potentially refer to the GetACP function (to obtain the current ANSI Code Page), and oem could refer to GetOEMCP, associated with the OEM code page. However, this is just a hypothesis. I could not correlate these settings with anything within this specific implant.

Inner map:



The outer map contains the ID, previously generated via the SHA1 hash, the type, from the configuration, and the data, which is the inner map data.



The registration beacon with the inner and outer maps is encrypted with the AES-128 key from the implant configuration (previously decrypted). This has to be known by the C2 to decrypt the registration information. This initial beacon includes the random session key.

After sending the registration beacon, the agent enters a loop to receive encrypted instructions from the C2. The commands received from the C2 are decrypted with the session key and deserialized with MessagePack unpacking. A type and an object are parsed from the response. The object is subsequently processed by a command dispatcher function.



Command Dispatcher

The object received from the C2 is subsequently deserialized and code, id, and data are parsed. The code determines which command code is executed by the agent, the id can be used to correlate commands and responses, and the data is additional data that a specific command may need, like a command line.



The agent supports the following commands:

Command	Description	Comment
0x1	Gets current working directory	Uses getcwd syscall
0x2	Changes working directory	Uses chdir syscall
0x3	Executes program with arguments	Unpacks 'program' and 'arguments'
0x4	Stops/exits	
0x5	File exfiltration	Allows for exfiltration of larger files. Chunked uploads
0x6	Writes files to disk	It can unzip delivered files
0x7/0x1B	Reads small files	Synchronous read of small files (<=1MB)
0x8	File copy	
0x9	File move/rename	
0xA	Creates a directory	Uses mkdir syscall
0xB	Deletes a file	Uses rm -rf
0xC	Directory Listing	Equivalent to ls -la
0xD	Process listing	Equivalent to ps aux

Command	Description	Comment
0xE	Kills a process by PID	Uses kill system call
0xF	ZIP compression	Does zip -r destination source
0x10	Screenshot	Returns “screenshot not supported”
0x11	Program Execution with a PTY	Could be used for interactive shell
0x12	Lists running jobs	Lists tasks run asynchronously
0x13	Cancels a job by ID	Uses the shutdown system call
0x15	Updates sleep and jitter	
0x16	Timestomping	Takes access time and modify time, uses utimensat syscall
0x17	Changes file permissions	Uses chmod syscall
0x18	Moves or renames a file	Takes old_path and new_path values
0x19	Creates a file	Creates a file with O_CREAT and O_WRONLY flags
0x1A	File write	Writes or overwrites a file
0x1F	Reverse Port forward/Pivot Tunnel	Accepts an address, protocol, and channel ID
0x20	Reverse Port forward/Pivot Tunnel Close	Closes the tunnel with a channel ID
0x23	Interactive program with PTY	Spawns program with full PTY
0x24	Kills interactive program with PTY	Uses shutdown syscall on socket and takes term_id
0x25	Sends Data Through Pivot	Bidirectional data relay into an existing 0x1F pivot tunnel

Command	Description	Comment
0x26	Connects to malware relay (a peer)	Potentially used to connect to 0x2D tunnel
0x27	Kills the connection to a peer	Likely related to peer connection from 0x26
0x29/0x2A	Returns “no agetty”	This looks like a feature that is not implemented yet
0x2D	Opens a listener	Binds to all interfaces. Accepts connections from any reachable host
0x2E	Closes a listener	Likely related to listeners opened via 0x2D
0x2F	Root directory enumeration	Minimal command that likely returns “/”
default	Returns “unknown”	

Overall, this looks like a low prevalence Linux implant with the capabilities that could be expected in malware used in a hands-on / targeted intrusion on a network. The capability to establish reverse tunnels, use the malware for pivoting, the implementation of port forwarding, and the multiple file extraction options suggest that this could potentially be used in a targeted intrusion.

I attempted to find similar samples, but did not find any. As always, if you recognize this as something previously reported or that you have seen yourself, feel free to reach out and share what you can.

IOCs

- Delivery URL: <https://zapier-logos.s3.amazonaws.com/gregbfdah.png>
- ELF: f264f04f597a2bdda372a27ae701c18d4036175b1eb8722db5f8531615b38788
- C2: [iot.981666\[.\]xyz](https://iot.981666.xyz):8080