Analysis of Node JS obfuscated malware: used to hijack Discord and YouTube accounts

POMMIER Thomas, KYPRIADIS Georgios, SANCHO Tom, BOUMEDAD Yanis

PoC Innovation - EPITECH

September 2023

Abstract: In the malware landscape, stealers are known for being easy to create and capable of causing significant harm to victims. They can be used to deceive YouTubers through phishing, tricking them into downloading a fake video game for a promotional review. This tactic allows for the theft of browser cookies, granting full access to YouTube channels and bypassing two-factor authentication.

In this research paper, we explore the characteristics of such malware and its goals. It involves static and dynamic analysis, breaking through Node JS obfuscation, and explaining the techniques used to slow down the reverse engineering process.

1. Introduction

Today, malicious software, often referred to as "malware," poses a significant and growing threat to digital security. These harmful programs can cause substantial damage by stealing sensitive data, compromising privacy, or disrupting online services. In this research paper, we focus our attention on 'cursed.exe,' a malware disguised as a video game. This malware was available on the black market and specialized in stealing information from Windows devices. It primarily spread through Discord channels, resulting in the hijacking of numerous Discord accounts and even some YouTube channels.

The focus of this paper is to provide a detailed analysis of the malware 'cursed.exe,' where we will present chronologically the methods and tools we employed. We'll start by explaining how we set up our environment for both dynamic and static analysis. Subsequently, we will delve into the various obfuscation techniques used to conceal the JavaScript code. We will then discuss the development of our custom deobfuscator, which played a crucial role in our in-depth analysis of the malware's inner workings. Finally, we will unveil the specific data that 'cursed.exe' secretly steals.

Through this research paper, our aim is to contribute to a broader understanding of stealers and Node.js reverse engineering.

2. Definitions

2.a. Static Analysis:

Static analysis is a method for inspecting software that does not require its actual execution. It involves examining the source code, decompiled binary, or other representations of the software to identify the nature of the malware. We can gather a lot of information about the executable architecture, dependencies used, and its original programming language.

However, this method has its limitations, it’s not recommended for obtaining a quick overview of what the malware essentially does. Furthermore, the process might be slowed down by code obfuscation. Nonetheless, for an in-depth analysis of malware, it is an obligatory step.

2.b. Dynamic Analysis:

Unlike the static approach, dynamic analysis involves the actual execution of malicious code in a secure environment, such as a Virtual Machine (VM), to closely observe all its actions and deduce its overall behavior during runtime. This allows us to capture system calls, monitor HTTP requests, and make guesses about its primary purpose without the need for in-depth analysis.

The process also includes debugging the executable and stepping through the malware's execution. This can reveal insights in dark spots, like string and control flow obfuscation, and is faster to observe compared to static analysis.

2.c. *Obfuscation / Deobfuscation:*

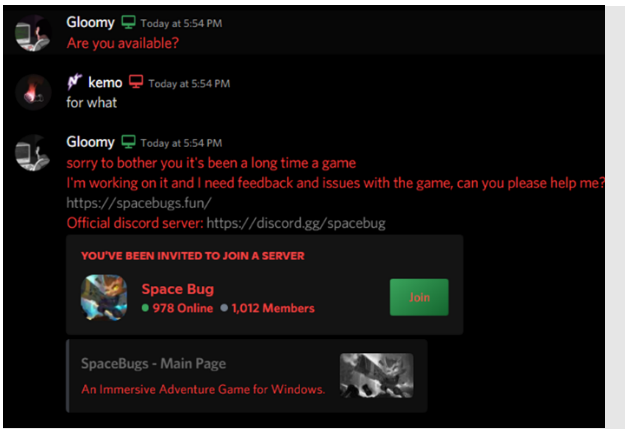
Obfuscation is the process of altering code to make it challenging to understand while maintaining its functionality. This practice is widely used in the software industry to safeguard proprietary code from reverse engineering and intellectual property theft. Nonetheless, obfuscated code can pose challenges when security analysts attempt to analyze malicious programs.

Subsequently, there is the deobfuscation process, which involves reversing code obfuscation. It employs various techniques and methods to convert obfuscated code back into a more human-readable and understandable form.

3. Analysis

3.a. Cursed.exe origins:

"Cursed.exe" typically arrives in the form of a message from someone on Discord. This "friend," after some social engineering, will ask your help in testing a game they claim to have been working on. They will then send you an apparently unsuspicious link along with convincing "proofs." (Fig. 1)



Gloomy Today at 5:54 PM
Are you available?

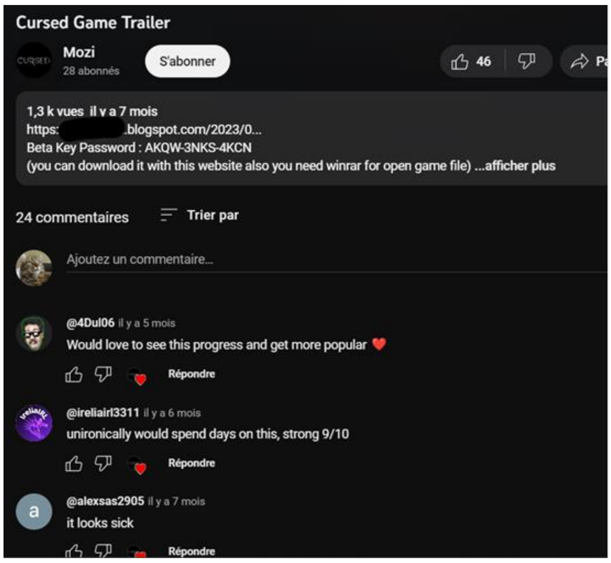
kemo Today at 5:54 PM
for what

Gloomy Today at 5:54 PM
sorry to bother you it's been a long time a game
I'm working on it and I need feedback and issues with the game, can you please help me?
https://spacebugs.fun/
Official discord server: https://discord.gg/spacebug

YOU'VE BEEN INVITED TO JOIN A SERVER
Space Bug
● 978 Online ● 1,012 Members
Join

SpaceBugs - Main Page
An Immersive Adventure Game for Windows.

Figure 1 : Infected account on discord giving a discord server and a website of a fake game



Cursed Game Trailer
Mozi
28 abonnés
S'abonner
46

1,3 k vues il y a 7 mois
https:[redacted].blogspot.com/2023/0...
Beta Key Password : AKQW-3NKS-4KCN
(you can download it with this website also you need winrar for open game file) ...afficher plus

24 commentaires Trier par

Ajoutez un commentaire...

@4Dul06 il y a 5 mois
Would love to see this progress and get more popular ❤️
Répondre

@ireliairl3311 il y a 6 mois
unironically would spend days on this, strong 9/10
Répondre

@alexsas2905 il y a 7 mois
it looks sick
Répondre

Figure 2: Cursed YouTube trailer with fake comments

The fake game even has a fake trailer (Fig. 2) stolen from a real game, using many bots to make it believable at first glance. The intentions are to rob your YouTube and Discord accounts.

3.b. Malware analysis lab:

For running and analyzing the malware, we used a very simple setup, using VirtualBox for virtualizing the operating system and the network.

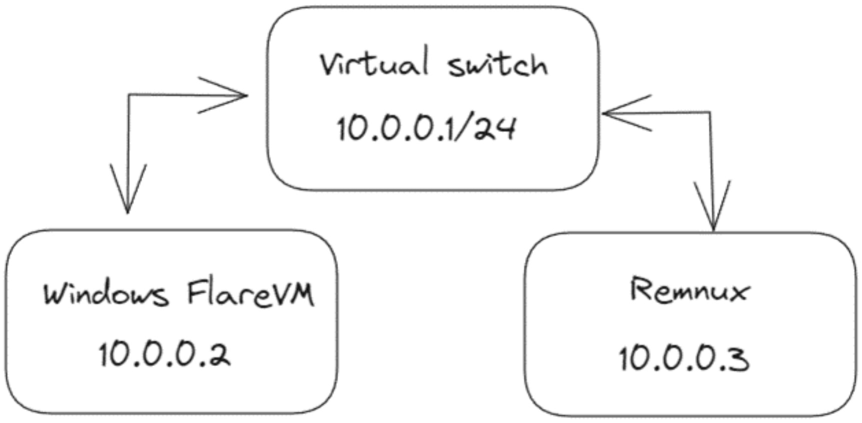


Figure 3: Schema of the environment used.

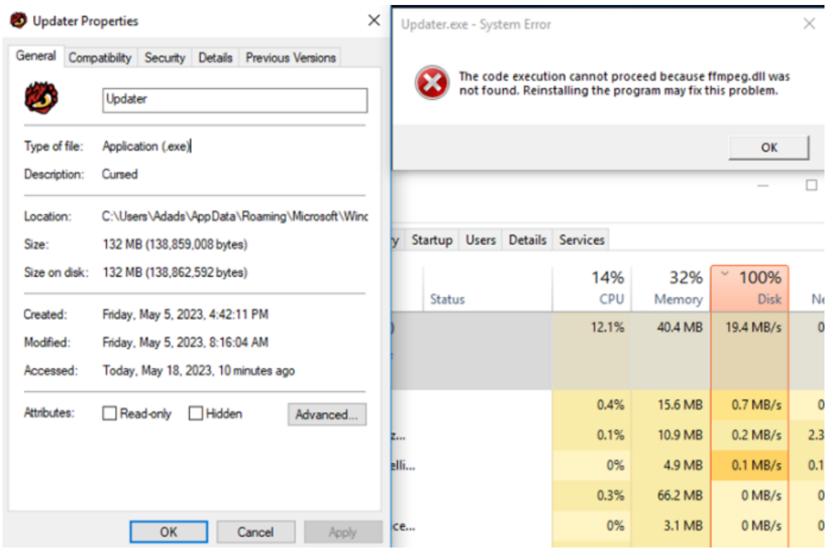
The Windows machine was used to run the malware and analyze it statically and dynamically using FlareVM tools, while the Remnux machine was mainly employed for networking and utilizing Linux-specific tools. (Fig. 3)

This setup allows us to stay completely isolated from the host machine by running both virtual machines in a private network. Then, we can use software like INetSim to simulate functioning websites, employing its DNS module to assign fake DNS settings. This deceives the virus into believing that it's online, as some of them kill themselves when they detect a protected environment.

3.c. Running the malware:

Once the lab setup was complete, the first action we took was running the malware, and the following events happened:

- There was no indication of infection after decompressing the .rar file.
- When executing the binary, no visible interface appeared, but the computer began to exhibit noticeable slowdown.
- In the Task Manager, numerous "Setup.exe" processes became apparent, and CPU usage remained stuck at 100%. (Fig. 4)
- After some time, a binary named "Updater.exe," with the same icon, requested firewall permissions.
- Upon each reboot, "Updater.exe" reported that "ffmpeg.dll" was missing.


Updater Properties

General | Compatibility | Security | Details | Previous Versions

Updater

Type of file: Application (.exe)
Description: Cursed

Location: C:\Users\Adads\AppData\Roaming\Microsoft\Winc
Size: 132 MB (138,859,008 bytes)
Size on disk: 132 MB (138,862,592 bytes)

Created: Friday, May 5, 2023, 4:42:11 PM
Modified: Friday, May 5, 2023, 8:16:04 AM
Accessed: Today, May 18, 2023, 10 minutes ago

Attributes: Read-only Hidden Advanced...

OK Cancel Apply

Updater.exe - System Error

The code execution cannot proceed because ffmpeg.dll was not found. Reinstalling the program may fix this problem.

OK

Status	14% CPU	32% Memory	100% Disk	Ne
	12.1%	40.4 MB	19.4 MB/s	0
	0.4%	15.6 MB	0.7 MB/s	0
	0.1%	10.9 MB	0.2 MB/s	2.3
	0%	4.9 MB	0.1 MB/s	0.1
	0.3%	66.2 MB	0 MB/s	0
	0%	3.1 MB	0 MB/s	0

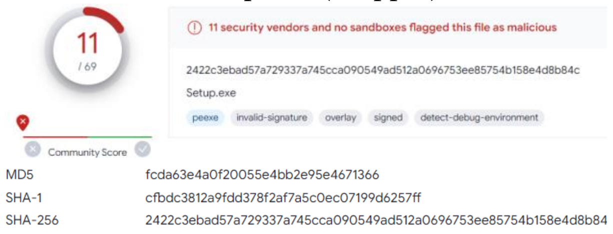
Figure 4: Malware running, properties of Update.exe, task manager and error message for ffmpeg.dll

Most of these events aren't particularly suspicious; high CPU usage can occur in normal system setups. The only alarming aspect is persistence, which is highly suspicious for a video game. It could potentially be justified for updates, but it’s a core part for most malwares, for example here it could be usen for stealing information in the long-term or running processes like crypto miners.

It’s not using any backdoor to keep the persistence hidden, Updater.exe is added to “C:\Users\{user}\AppData\Roaming\Microsoft\Windows\Start Menu\Programs\Startup” and Windows handles the rest.

3.c. Anti-virus:

VirusTotal of Setup.exe (dropper):


11 / 69

11 security vendors and no sandboxes flagged this file as malicious

2422c3ebad57a729337a745cca090549ad512a0696753ee85754b158e4d8b84c

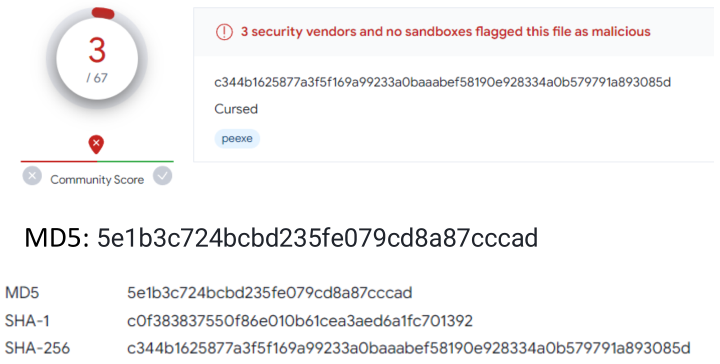
Setup.exe

peexe | invalid-signature | overlay | signed | detect-debug-environment

MD5	fcda63e4a0f20055e4bb2e95e4671366
SHA-1	cfbdc3812a9fdd378f2af7a5c0ec07199d6257ff
SHA-256	2422c3ebad57a729337a745cca090549ad512a0696753ee85754b158e4d8b84

MD5: fcda63e4a0f20055e4bb2e95e4671366

VirusTotal of Cursed.exe (dropped):


3 / 67

3 security vendors and no sandboxes flagged this file as malicious

c344b1625877a3f5f169a99233a0baaabef58190e928334a0b579791a893085d

Cursed

peexe

MD5: 5e1b3c724bcbd235fe079cd8a87cccad

MD5	5e1b3c724bcbd235fe079cd8a87cccad
SHA-1	c0f383837550f86e010b61cea3aed6a1fc701392
SHA-256	c344b1625877a3f5f169a99233a0baaabef58190e928334a0b579791a893085d

Even though the binaries have been spread widely anti-virus still seem to struggle with them, Setup.exe have been flagged by some popular anti-virus but it’s still bad for a binary in the wild for 1 year. Cursed.exe does not have any convincing detection.

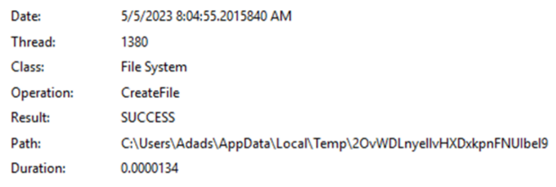
3.d. Basic dynamic analysis:

1) Syscalls analysis

Given the hints we had about the program's behavior, such as the presence of another binary called "Updater.exe," the most efficient approach was to initiate basic dynamic analysis using Sysinternals tools (Microsoft's internal tools).

We used Process Monitor, a program that allows us to observe all system calls on the system and filter them.

We then executed the malware once more on a clean VM with Process Monitor running. By applying specific filters, we could observe the following: (Fig. 5)



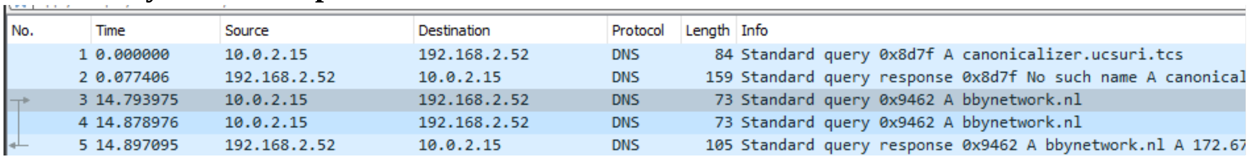
Date:	5/5/2023 8:04:55.2015840 AM
Thread:	1380
Class:	File System
Operation:	CreateFile
Result:	SUCCESS
Path:	C:\Users\Adads\AppData\Local\Temp\2OvWDLnyellvHXDxkpnFNUlbel9
Duration:	0.0000134

Figure 5: Process monitor output from cursed.exe

“Setup.exe” runs a “CreateFile” syscall in the path: **“C:\Users\Adads\AppData\Local\Temp\20vWDLnyellvHXDxkpnFNUlbel9”**. In it, we can finally see Cursed.exe in person with bunch of dynamic libraries. All those libraries are very common for video games which can make us think this is not really malware.

2) <u>Basic network analysis:</u>

The first thing to know is which port the binary uses, for this we use netstat and identified port 5353 as the only one used, with this data we can use the advantage of our virtual lab, we can run our network emulator (INetSim) and start to analyze the port with Wireshark. (Fig. 6)



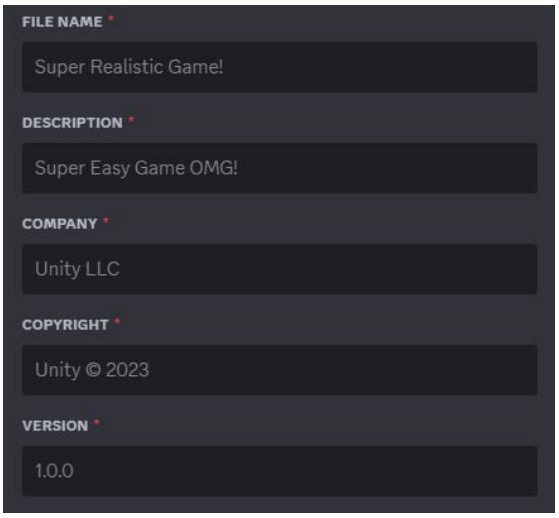
```
No.   Time        Source         Destination    Protocol  Length Info
    1 0.000000    10.0.2.15      192.168.2.52   DNS           84 Standard query 0x8d7f A canonicalizer.ucsuri.tcs
    2 0.077406    192.168.2.52   10.0.2.15      DNS          159 Standard query response 0x8d7f No such name A canonical
    3 14.793975   10.0.2.15      192.168.2.52   DNS           73 Standard query 0x9462 A bbynetwork.nl
    4 14.878976   10.0.2.15      192.168.2.52   DNS           73 Standard query 0x9462 A bbynetwork.nl
    5 14.897095   192.168.2.52   10.0.2.15      DNS          105 Standard query response 0x9462 A bbynetwork.nl A 172.67
```

Figure 6: Wireshark output from cursed.exe showing DNS requests

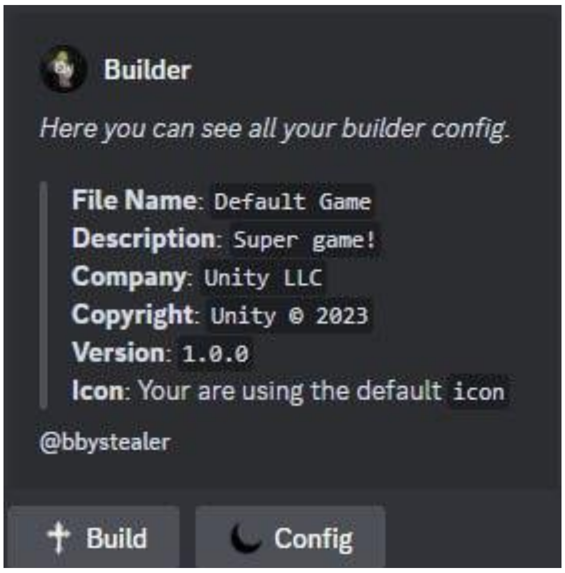
The software sends DNS query to a domain name, if we do a little search, a reddit post comes up mentioning scam game “Damned” with the same methods as Cursed.

A telegram also shows up, we can see developer updates, a music video, ads for the stealer. There is a huge community of 4600 followers behind it. (Fig. 7)

They also have a discord for customers, it’s where clients can build the malware for 45$, customers are all identifiable with their “Customer” role.



FILE NAME *	Super Realistic Game!
DESCRIPTION *	Super Easy Game OMG!
COMPANY *	Unity LLC
COPYRIGHT *	Unity © 2023
VERSION *	1.0.0



Builder

Here you can see all your builder config.

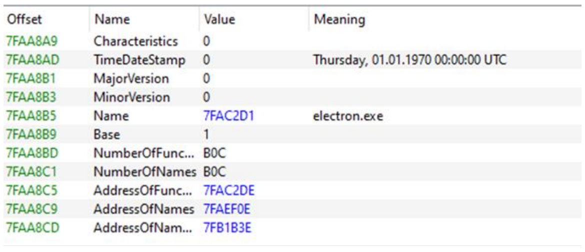
File Name: `Default Game`
Description: `Super game!`
Company: `Unity LLC`
Copyright: `Unity © 2023`
Version: `1.0.0`
Icon: Your are using the default `icon`

@bbystealer

Figure 7: Builder for clients of the malware

3.e. Quick static analysis on Updater.exe:

Since nothing happens when we launch the program, we needed to do a bit of static analysis to get some hints. The easiest thing we can do is check for suspicious information in PE header (portable executable format on Windows): (Fig. 8)



Offset	Name	Value	Meaning
7FAA8A9	Characteristics	0	
7FAA8AD	TimeDateStamp	0	Thursday, 01.01.1970 00:00:00 UTC
7FAA8B1	MajorVersion	0	
7FAA8B3	MinorVersion	0	
7FAA8B5	Name	7FAC2D1	electron.exe
7FAA8B9	Base	1	
7FAA8BD	NumberOfFunc...	B0C	
7FAA8C1	NumberOfNames	B0C	
7FAA8C5	AddressOfFunc...	7FAC2DE	
7FAA8C9	AddressOfNames	7FAEF0E	
7FAA8CD	AddressOfNam...	7FB1B3E	

Figure 8: Updater.exe in PE Studio (NT Headers exports)

Quickly, we saw the mention of electron.exe in the exports. If we combine this information with the size of the binary (135mb) it's very probable that the application is written in JavaScript using Electron.

This fact makes all usual software for static analyzers like IDA or dynamic ones like x64dbg totally inefficient since they work only on compiled binaries. Also, the fact that it might just be a game becomes way less probable.

4. Deobfuscator:

4.a. Preliminaries:

Once we discovered a .asar file within the malware's .exe, it became evident that this was not a compiled binary but rather a JavaScript desktop application. ASAR (Atom Shell Archive) files in Node JS serve the purpose of packaging application files and assets into a single compressed archive [1]. This approach facilitates the distribution of Electron-based applications in the ".exe" format. That means the malware don't need the Node JS interpreter installed somewhere, since ASAR handles it. Node JS malwares are uncommon since it requires the Node JS runtime installed, but bundling with ASAR makes it portable to every windows device.

To access the files contained within this .asar file, we had to install "asar" via NPM (Node Package Manager) [2]. After extracting with the asar module, we found multiple files:

- Node_modules
- Index.html
- Index.js
- Package.json
- Package-lock.json

These files are obviously from a Node JS/Electron application. After inspecting "index.js" and "index.html," it became apparent that they had been obfuscated. Our first reflex was to test existing JavaScript deobfuscators to determine if the obfuscator used was supported. We experimented with tools like "restringer, [3]", "webcrack [4]", "de4js [5]" but none of them seemed to show any useful results.

It failed because the Abstract Syntax Tree (AST) parser couldn't successfully parse the code. This was primarily due to the malware employing ECMAScript 2020, while native Esprima (the AST parser for these deobfuscators) only supports ECMAScript 2019 [6].
An Abstract Syntax Tree (AST) helps to break down the code into a tree of nodes. This tree structure is essential for compilers, as it utilized in the translation of human-readable code into machine code. In JavaScript, ASTs are primarily utilized by JavaScript libraries such as jQuery (Esprima is from this library [7]). In the context of deobfuscation, the tree data structure of the AST simplifies the process of replacing obfuscated code via nodes.
Fortunately, more recent AST parsers which support higher version of ECMAScript are available. We tried to replace the old Esprima dependency from the existing deobfuscators, but we had the same disappointing results. To understand the code, we needed to make our own tools.

In the git repo [8] included with this research paper, there are two deobfuscators: one for index.html and another for index.js. Programming a deobfuscator requires time and analyzing skills to determine how the code translates in real time. Therefore, we needed a debugger.

4.b Obfuscated Code Analysis:

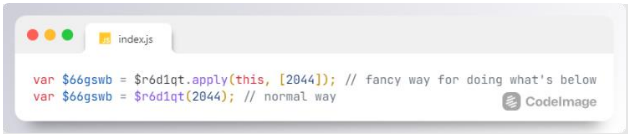
To debug Node.js code, there's no need for specific debuggers. We used the VSCode integrated debugger, which supports Electron [9]. Luckily, JavaScript is easy to debug, and the malware doesn't make any eval function calls. This means that debugging the code can directly reveal internal strings, either through hovering over a variable or using console logs. However, this method is quite time-consuming and can be tedious.

After analyzing, here is categorized methods this obfuscator uses:

1. String Obfuscation
   Almost every single string has been obfuscated. All of them use the indexer and decryptor methods to be translated (see below).

2. Name Obfuscation
   Every variable and function names are obfuscated. This makes the code harder to read since we don't know the original purpose of the variables. However, if we know their content, we can guess them.

3. Function Call Obfuscation
   In JavaScript, a function may be called with method expressions instead of the traditional (). Example: (Fig. 9)

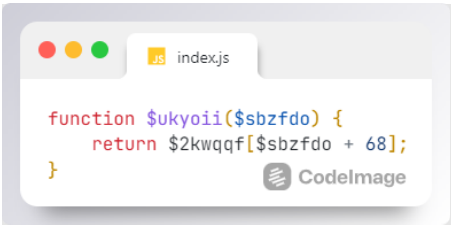


```
var $66gswb = $r6d1qt.apply(this, [2044]); // fancy way for doing what's below
var $66gswb = $r6d1qt(2044); // normal way
```

Figure 9: Example of function call obfuscation

4. Indexers
   The program contains JavaScript arrays full of data (encrypted strings, random numbers). These indexers are “proxy functions” to access the arrays. This data will be used by **Decryptors** to translate a Literal (= string, number, Boolean, null etc…).

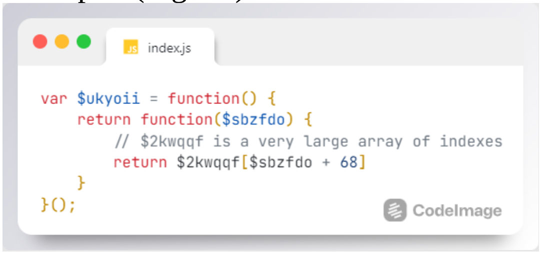
   a. Static
      Static indexers are defined once but won't be redefined after, detecting them is easy since in this malware, they are “function” defined (instead of “arrowed” function: () =>). (Fig. 10)



```
function $ukyoii($sbzfdo) {
    return $2kwqqf[$sbzfdo + 68];
}
```

Figure 10: Example of a static indexer function

5. Dynamic
   Dynamic indexers are defined multiple times through the workflow. This is more challenging to handle since they use the same variable name multiple times (depending on the scope)
   Example: (Fig. 11)

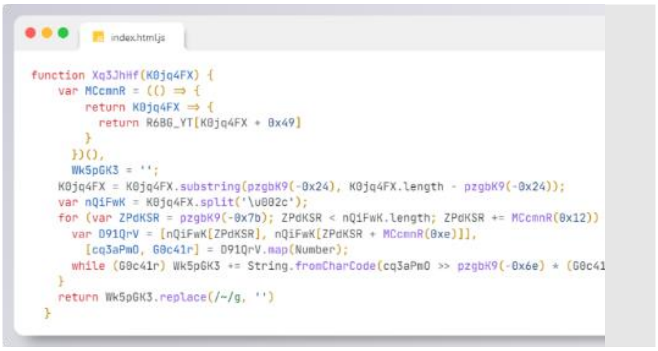


```
var $ukyoii = function() {
    return function($sbzfdo) {
        // $2kwqqf is a very large array of indexes
        return $2kwqqf[$sbzfdo + 68]
    }
}();
```

Figure 11: Example of a dynamic indexer function

6. Decryptors

   a. Static
      The decryptors here are not particularly hidden and are defined only once. They utilize an index to retrieve the string from an array of encrypted strings. In fact, indexers and decryptors are frequently used together. In index.js, we identified and used two string decryptor functions that employ different decryption methods. The challenge is to make sure the input index is correct via prior deobfuscation.
      Example: (no need to look in the details) (Fig. 12)



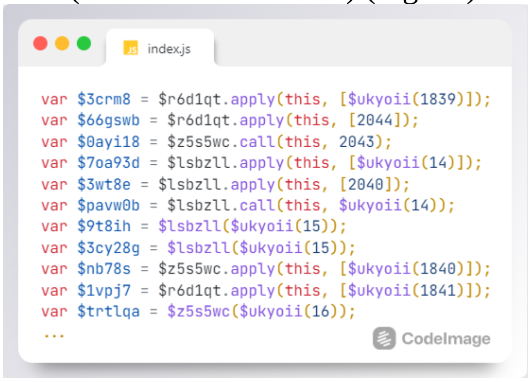
```
function Xq3JhHf(K0jq4FX) {
    var MCcmnR = (() => {
        return K0jq4FX => {
          return R6BG_YT[K0jq4FX + 0x49]
        }
      })(),
      Wk5pGK3 = '';
    K0jq4FX = K0jq4FX.substring(pzgbK9(-0x24), K0jq4FX.length - pzgbK9(-0x24));
    var nQiFwK = K0jq4FX.split('\u002c');
    for (var ZPdKSR = pzgbK9(-0x7b); ZPdKSR < nQiFwK.length; ZPdKSR += MCcmnR(0x12))
      var D91QrV = [nQiFwK[ZPdKSR], nQiFwK[ZPdKSR + MCcmnR(0xe)]],
        [cq3aPm0, G0c41r] = D91QrV.map(Number);
      while (G0c41r) Wk5pGK3 += String.fromCharCode(cq3aPm0 >> pzgbK9(-0x6e) * (G0c41
    }
    return Wk5pGK3.replace(/~/g, '')
  }
```

Figure 12: Example of a static decryptor function

   b. Dynamic
      The functions are not defined multiple times (in contrary of dynamic indexers), but the process of decrypting content may have side effects. For example: a string is deobfuscated, it pushed that decrypted string inside another array which is read elsewhere outside the function.
      Another one is to rotate an out-of-scope array multiple times before using it, with the loop condition obfuscated.

7. Proxy

   a. Variables
      The obfuscator uses a lot of variables. The vast majority contains pieces of string to concatenate with other variables to create a string. This purpose is to concatenate the same piece of string multiple times in different parts of the code. The example below shows what it looks like: (1300 lines like this...) (Fig. 13)

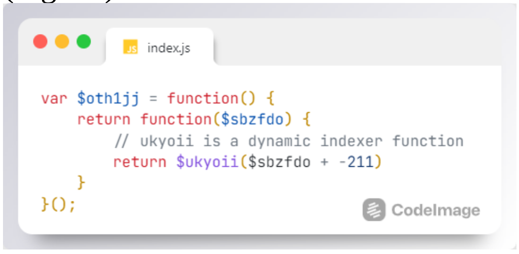


```
var $3crm8 = $r6d1qt.apply(this, [$ukyoii(1839)]);
var $66gswb = $r6d1qt.apply(this, [2044]);
var $0ayi18 = $z5s5wc.call(this, 2043);
var $7oa93d = $lsbzll.apply(this, [$ukyoii(14)]);
var $3wt8e = $lsbzll.apply(this, [2040]);
var $pavw0b = $lsbzll.call(this, $ukyoii(14));
var $9t8ih = $lsbzll($ukyoii(15));
var $3cy28g = $lsbzll($ukyoii(15));
var $nb78s = $z5s5wc.apply(this, [$ukyoii(1840)]);
var $1vpj7 = $r6d1qt.apply(this, [$ukyoii(1841)]);
var $trtlqa = $z5s5wc($ukyoii(16));
...
```

Figure 13: Example of a batch of proxy variables

   b. Functions

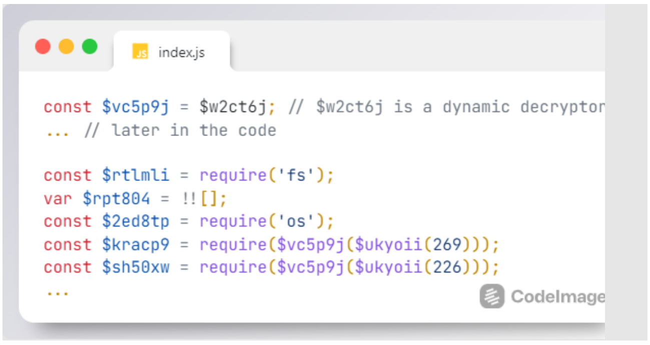
      i. Redefinition
         A function is defined via the arrow operator. It calls to a dynamic or static indexer previously defined with a different index. Example: (Fig. 14)



```
var $oth1jj = function() {
    return function($sbzfdo) {
        // ukyoii is a dynamic indexer function
        return $ukyoii($sbzfdo + -211)
    }
}();
```

Figure 14: Example of a dynamic function indexer, which is a redefinition proxy.

ii. Aliasing (Variable)
Simple yet effective function aliasing (function pointer) via a variable. Example: (Fig. 15)

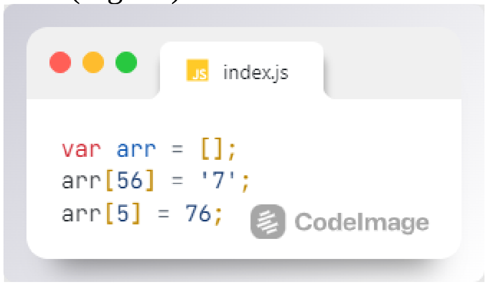


index.js

```
const $vc5p9j = $w2ct6j; // $w2ct6j is a dynamic decryptor
... // later in the code

const $rtlmli = require('fs');
var $rpt804 = !![];
const $2ed8tp = require('os');
const $kracp9 = require($vc5p9j($ukyoii(269)));
const $sh50xw = require($vc5p9j($ukyoii(226)));
...
```

Figure 15: Example of function aliasing ($vc5p9j)

8. Variadic function obfuscation
JavaScript is a very permissive language. Unlike C/Python arrays or C++ vectors, JavaScript arrays don't require size definition in advance, and there's no need to reallocate memory to change their size. JavaScript arrays are secret objects that support indexes, allowing you to perform operations like: (Fig. 16)

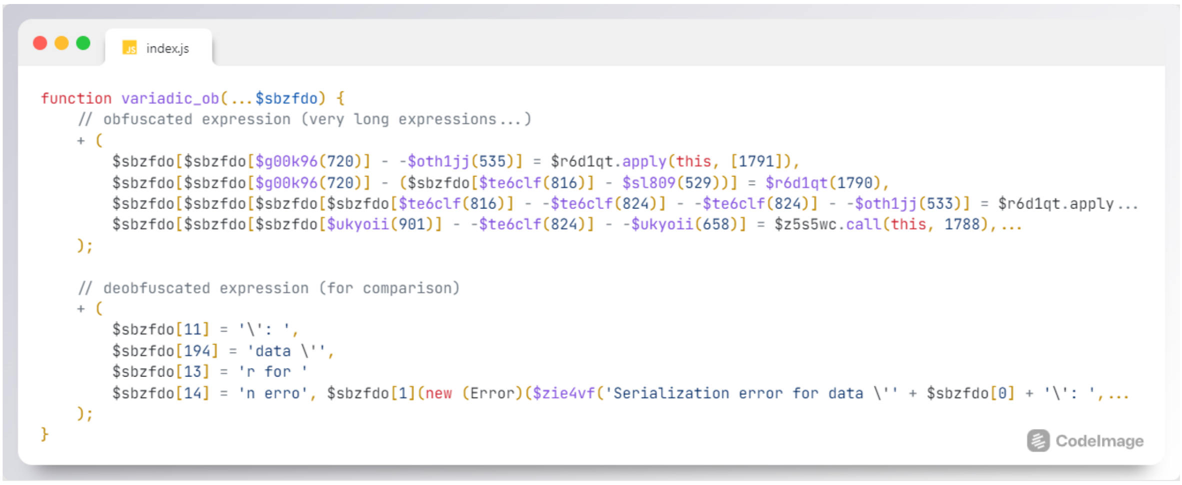


index.js

```
var arr = [];
arr[56] = '7';
arr[5] = 76;
```

Figure 16: JavaScript illegal array indexing

This type of code above is forbidden by coding style standards. However, the obfuscator is using this possibility to hide function parameters.
Instead of a declared array variable like in the example, it uses a variadic function variable which is the same type. Normally, using a variadic function is useful when you don't know the number of arguments in advance. The obfuscator benefits from this functionality for hiding the parameters via multiple series of indexers.
Example: (Fig. 17)



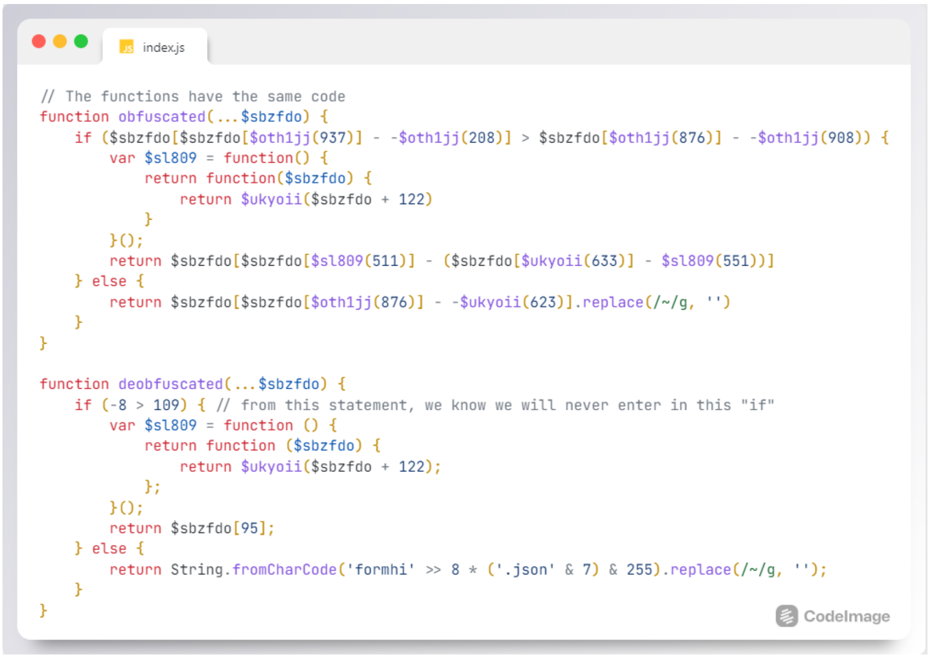
index.js

```
function variadic_ob(...$sbzfdo) {
    // obfuscated expression (very long expressions...)
    + (
        $sbzfdo[$sbzfdo[$g00k96(720)] - -$oth1jj(535)] = $r6d1qt.apply(this, [1791]),
        $sbzfdo[$sbzfdo[$g00k96(720)] - ($sbzfdo[$te6clf(816)] - $sl809(529))] = $r6d1qt(1790),
        $sbzfdo[$sbzfdo[$sbzfdo[$sbzfdo[$te6clf(816)] - -$te6clf(824)] - -$te6clf(824)] - -$oth1jj(533)] = $r6d1qt.apply...
        $sbzfdo[$sbzfdo[$sbzfdo[$ukyoii(901)] - -$te6clf(824)] - -$ukyoii(658)] = $z5s5wc.call(this, 1788),...
    );

    // deobfuscated expression (for comparison)
    + (
        $sbzfdo[11] = '\': ',
        $sbzfdo[194] = 'data \'',
        $sbzfdo[13] = 'r for '
        $sbzfdo[14] = 'n erro', $sbzfdo[1](new (Error)($zie4vf('Serialization error for data \'' + $sbzfdo[0] + '\': ',...
    );
}
```

Figure 17: Example of variadic function obfuscation

Sometimes, the expression is so long that the number of characters is greater than *editor.stopRenderingLineAfter* in VSCode (which is above 10,000 characters). In this example (Fig. 17), it was a small expression (+ beautified).

9. Workflow obfuscation
Every obfuscator comes with a workflow obfuscation method. It wants to confuse the reverse engineer with obfuscated conditional expressions that in fact, will never be evaluated.
Example: (Fig. 18)


index.js

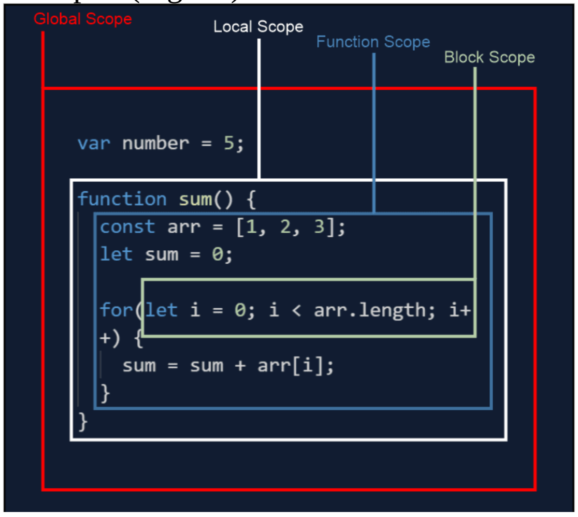
```
// The functions have the same code
function obfuscated(...$sbzfdo) {
    if ($sbzfdo[$sbzfdo[$oth1jj(937)] - -$oth1jj(208)] > $sbzfdo[$oth1jj(876)] - -$oth1jj(908)) {
        var $sl809 = function() {
            return function($sbzfdo) {
                return $ukyoii($sbzfdo + 122)
            }
        }();
        return $sbzfdo[$sbzfdo[$sl809(511)] - ($sbzfdo[$ukyoii(633)] - $sl809(551))]
    } else {
        return $sbzfdo[$sbzfdo[$oth1jj(876)] - -$ukyoii(623)].replace(/~/g, '')
    }
}

function deobfuscated(...$sbzfdo) {
    if (-8 > 109) { // from this statement, we know we will never enter in this "if"
        var $sl809 = function () {
            return function ($sbzfdo) {
                return $ukyoii($sbzfdo + 122);
            };
        }();
        return $sbzfdo[95];
    } else {
        return String.fromCharCode('formhi' >> 8 * ('.json' & 7) & 255).replace(/~/g, '');
    }
}
```

Figure 18: Example of workflow obfuscation

It's challenging to determine whether the “if” branch will execute or not. The deobfuscator demonstrates in the example (Fig.18) that the flow will never run in it.

10. Global Function Obfuscation
    The program contains console.logs function calls, but we can’t see any at first glance. Since console is a global object (you don’t need to import it with **require()**), the obfuscator uses the object **global** and a function to translate everything.
    In fact, console.log may be written like this: global[“console”][“log”] with “console” and “log” as encrypted strings.

11. Scoping:
    The malware makes use of scoping to make it harder to translate variables. Scopes help to separate variables from the rest of the code like in this example: (Fig. 19)



```
var number = 5;

function sum() {
  const arr = [1, 2, 3];
  let sum = 0;

  for(let i = 0; i < arr.length; i++) {
    sum = sum + arr[i];
  }
}
```

Figure 19: Example of scoping in JavaScript [10]

In JavaScript, there are multiple scopes but 3 are the important ones: Global, function and block. But we can consider 2 scopes if we consider global as a function call (which our deobfuscator considers).
The “var” keyword only respects the function scope, where it only triggers when the program enters into a function. This keyword is very permissive, even too much for many programmers. To remediate this, recent versions of JavaScript introduced **block** scopes which are respected by

"let" and "const" variable types. Generally, a block scope is inside curly brackets (function included), example: if statements, try/catch, for loops etc…

12. Object Obfuscation (unsupported by the deobfuscator)
    Our deobfuscator uses decrypted strings to map content, but this is unsupported because both index.js and index.html rarely rely on this type of obfuscation. This approach may result in many false positives when detecting the obfuscation on original objects.

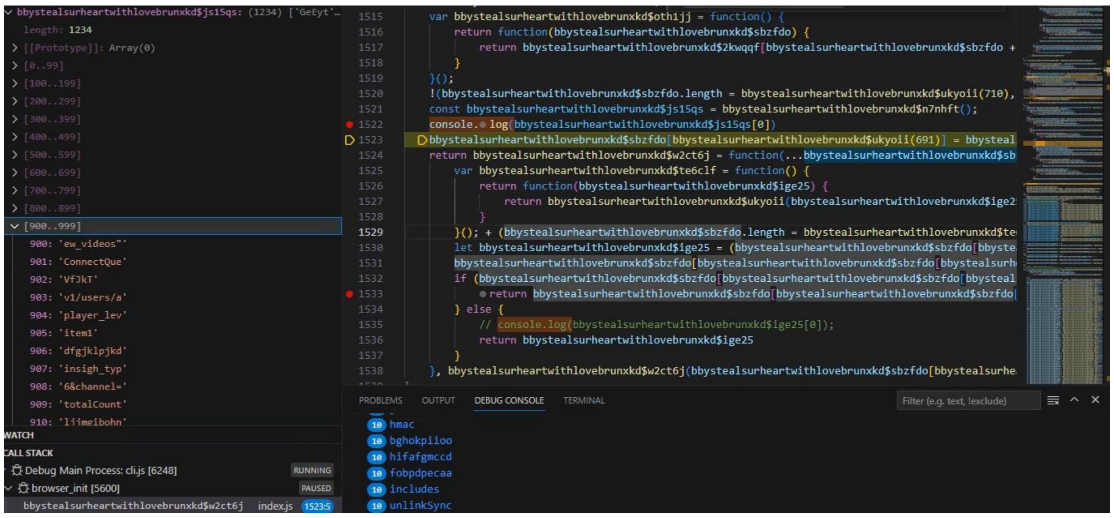
Some of these obfuscation techniques, such as function call obfuscations, static decryptors, and indexer functions, are easy to handle and do not pose significant challenges. However, in the case of dynamic redefinition and proxying, if the deobfuscator fails to detect them, the whole translation process may fail. A wrong index can cause a decryptor function to throw an error since it may be working with the wrong types. For instance, a variable can be originally a string and, at some point, become a function. Failing to capture this change may crash the deobfuscator.

4.c Deobfuscation methods:

This research paper does not delve into the inner workings of the deobfuscator. Instead, this section briefly explains how it handles the obfuscation techniques described above:

1. Parsing the obfuscated Code
   We use **exprima-next** [11] to convert the code into an AST and **escodegen** [12] to compile the AST into readable code. We traverse the tree using a postorder traversal method to ensure that nested obfuscation techniques are handled properly.

2. Encrypted Data arrays
   The arrays are stored directly inside a JavaScript file so that both the static and dynamic functions can access them. Some of these arrays have been modified at runtime. This means we manually extracted their values from the debugger after they were defined. Here in an example of extracting data via the VSCode debugger: (Fig. 20)



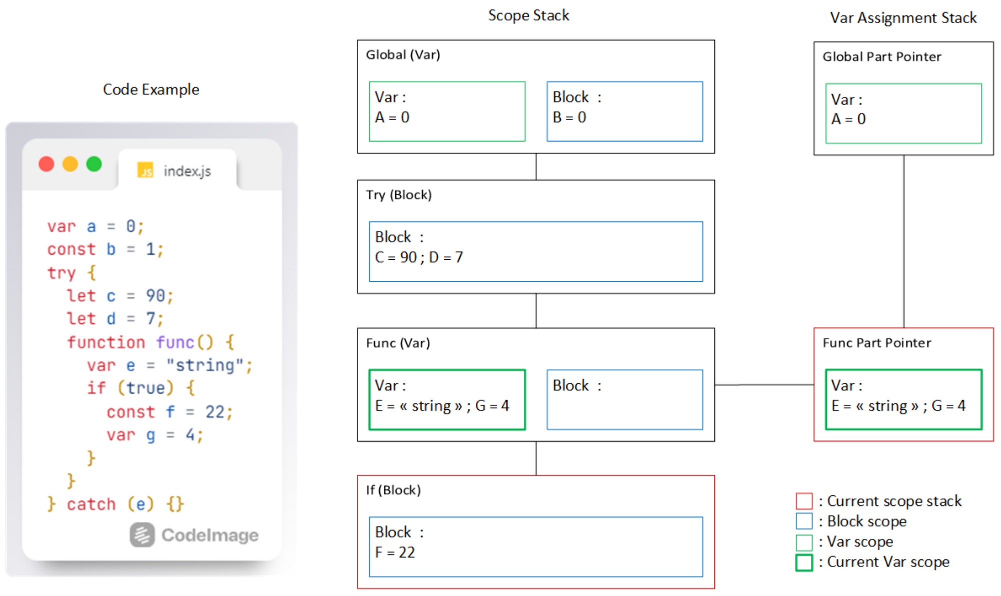
```
bbystealsurheartwithlovebrunxkd$js15qs: (1234) ['GeEyt'…
  length: 1234
> [[Prototype]]: Array(0)
> [0..99]
> [100..199]
> [200..299]
> [300..399]
> [400..499]
> [500..599]
> [600..699]
> [700..799]
> [800..899]
v [900..999]
    900: 'ew_videos"'
    901: 'ConnectQue'
    902: 'VfJkT'
    903: 'v1/users/a'
    904: 'player_lev'
    905: 'item1'
    906: 'dfgjklpjkd'
    907: 'insigh_typ'
    908: '6&channel='
    909: 'totalCount'
    910: 'ljimgibohn'
WATCH
CALL STACK
  Debug Main Process: cli.js [6248]          RUNNING
  browser_init [5600]                        PAUSED
    bbystealsurheartwithlovebrunxkd$w2ct6j   index.js 1523:5
```

```
var bbystealsurheartwithlovebrunxkd$oth1jj = function() {
    return function(bbystealsurheartwithlovebrunxkd$sbzfdo) {
        return bbystealsurheartwithlovebrunxkd$2kwqqf[bbystealsurheartwithlovebrunxkd$sbzfdo +
    }
}();
!(bbystealsurheartwithlovebrunxkd$sbzfdo.length = bbystealsurheartwithlovebrunxkd$ukyoii(710),
const bbystealsurheartwithlovebrunxkd$js15qs = bbystealsurheartwithlovebrunxkd$n7nhft();
console.log(bbystealsurheartwithlovebrunxkd$js15qs[0])
bbystealsurheartwithlovebrunxkd$sbzfdo[bbystealsurheartwithlovebrunxkd$ukyoii(691)] = bbysteal
return bbystealsurheartwithlovebrunxkd$w2ct6j = function(...bbystealsurheartwithlovebrunxkd$sb
    var bbystealsurheartwithlovebrunxkd$te6clf = function() {
        return function(bbystealsurheartwithlovebrunxkd$ige25) {
            return bbystealsurheartwithlovebrunxkd$ukyoii(bbystealsurheartwithlovebrunxkd$ige2
        }
    }(); + (bbystealsurheartwithlovebrunxkd$sbzfdo.length = bbystealsurheartwithlovebrunxkd$te
    let bbystealsurheartwithlovebrunxkd$ige25 = (bbystealsurheartwithlovebrunxkd$sbzfdo[bbyste
    bbystealsurheartwithlovebrunxkd$sbzfdo[bbystealsurheartwithlovebrunxkd$sbzfdo[bbystealsurh
    if (bbystealsurheartwithlovebrunxkd$sbzfdo[bbystealsurheartwithlovebrunxkd$sbzfdo[bbysteal
        return bbystealsurheartwithlovebrunxkd$sbzfdo[bbystealsurheartwithlovebrunxkd$sbzfdo[
    } else {
        // console.log(bbystealsurheartwithlovebrunxkd$ige25[0]);
        return bbystealsurheartwithlovebrunxkd$ige25
    }
}, bbystealsurheartwithlovebrunxkd$w2ct6j(bbystealsurheartwithlovebrunxkd$sbzfdo[bbystealsurhe
```

```
10 hmac
10 bghokpiioo
10 hifafgmccd
10 fobpdpecaa
10 includes
10 unlinkSync
```

Figure 20: VSCode debugger with the variable explorer while debugging cursed.exe

3. Scoping



```javascript
var a = 0;
const b = 1;
try {
  let c = 90;
  let d = 7;
  function func() {
    var e = "string";
    if (true) {
      const f = 22;
      var g = 4;
    }
  }
} catch (e) {}
```

Figure 21: Scoping Class schema used by the deobfuscator

To simulate scoping, we created a Scope class (Fig. 21) that works as follows: we have a scope stack (array) that emulates the Pile data structure. It contains both ”var” and ”block” scope, and all variables are placed in the appropriate category. When there's a new scope (such as an if statement, function definition, or try-catch block), a new stack frame is created for future variables assignments. The class also has a “Var Assignment Stack” that contains pointers to the last "var" scope category. This approach is more optimized since ”var” type variables are rarer than block variables, and we directly link the object to avoid having to loop through the main "Scope Stack” array during the assignment process.

4. Indexers

a. Static
We directly add the result of the function, which was literally copied into the deobfuscator, into an AST node.

b. Dynamic
The deobfuscator detects if it's a variable declaration (all dynamic indexers are variable functions) and checks if it returns something from an encrypted data array. If it does, we define the function in the scope.

5. Decryptors

a. Static
Like static indexers, we store the function definition inside the deobfuscator and call it to translate a string via an index.

b. Dynamic
We define the decryptor as it is in the deobfuscator. Then, we detect which side effects occur and create an environment when during deobfuscator runtime, the function applies the side effect correctly (see the $lsbzll function): [13]

6. Binary Expressions
Binary expressions involve using two literals in an operation (+, -, >, <=, etc.). For example: 1 + 2. The deobfuscator automatically evaluates these expressions with error handling. Therefore, if two strings are added, they are already concatenated in the

final output since the deobfuscator only allows both Literal values as parameters.

7. Proxy

   a. Variables
   Our Proxy system detects if a variable is defined and puts its name and result inside the scope. If the variable is identified during a binary expression, the variable automatically gets replaced by its content.

   b. Functions

      i. Redefinition
      This has similar code from the dynamic indexers since it uses the same technique. However, we first check if it calls another function. If the function call is a proxy function (which is always the case), the function calls the proxied function and directly returns the correct value. The function definition is then added to the scope for future use.

      ii. Variable
      If a variable assigns to a function proxy variable, it becomes a variable function proxy. In such cases, we put the proxy function inside the scope with the "pr_" prefix and add it to the scope. Once a CallExpression node is detected, the scope class checks if the function name is a proxy. If it is, it calls directly the proxied function.

8. Variadic obfuscation
   This method of obfuscation always starts with expressions that are ExpressionStatement nodes, such as !(), ~(), +()... We create an array with the same name as the variadic variable in the scope. Then, we perform a post-order traversal to replace the values one by one with the scoped array through node replacement.

9. Workflow obfuscation
   Thanks to everything above, workflow obfuscation is easily detected by "always false" if statements.

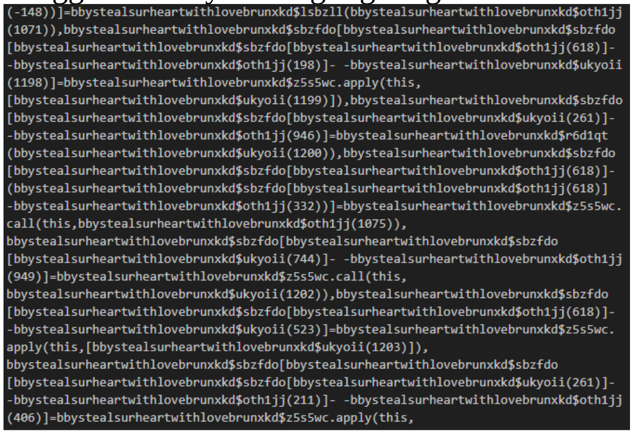
10. Global Function Obfuscation
    There is a function that hides global objects (such as console, Object, and module) and uses an index to translate the correct value. We simply call the function and use the result. In the deobfuscator output, since it's a CallExpression node by default and for the sake of saving time, we replace the function name with the return value. However, this means that the outputted deobfuscation result cannot be run due to a syntax error.

5. Analyze Post Deobfuscation:

5.a. Running the deobfuscator:

Once we implemented our deobfuscator, we can finally start to use it to analyze our program. If you want to test the deobfuscator, the readme indicates the steps.

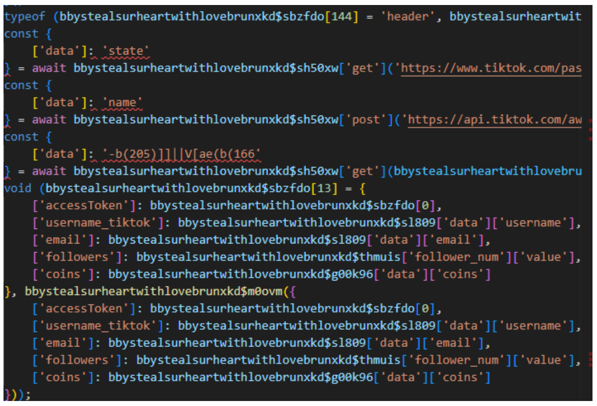
Here is the original version (Fig. 22), the code is highly obfuscated, the editor even struggles with syntax highlighting.



```
(-148))]=bbystealsurheartwithlovebrunxkd$lsbzll(bbystealsurheartwithlovebrunxkd$oth1jj
(1071)),bbystealsurheartwithlovebrunxkd$sbzfdo[bbystealsurheartwithlovebrunxkd$sbzfdo
[bbystealsurheartwithlovebrunxkd$sbzfdo[bbystealsurheartwithlovebrunxkd$oth1jj(618)]-
-bbystealsurheartwithlovebrunxkd$oth1jj(198)]- -bbystealsurheartwithlovebrunxkd$ukyoii
(1198)]=bbystealsurheartwithlovebrunxkd$z5s5wc.apply(this,
[bbystealsurheartwithlovebrunxkd$ukyoii(1199)]),bbystealsurheartwithlovebrunxkd$sbzfdo
[bbystealsurheartwithlovebrunxkd$sbzfdo[bbystealsurheartwithlovebrunxkd$ukyoii(261)]-
-bbystealsurheartwithlovebrunxkd$oth1jj(946)]=bbystealsurheartwithlovebrunxkd$r6d1qt
(bbystealsurheartwithlovebrunxkd$ukyoii(1200)),bbystealsurheartwithlovebrunxkd$sbzfdo
[bbystealsurheartwithlovebrunxkd$sbzfdo[bbystealsurheartwithlovebrunxkd$oth1jj(618)]-
(bbystealsurheartwithlovebrunxkd$sbzfdo[bbystealsurheartwithlovebrunxkd$oth1jj(618)]
-bbystealsurheartwithlovebrunxkd$oth1jj(332))]=bbystealsurheartwithlovebrunxkd$z5s5wc.
call(this,bbystealsurheartwithlovebrunxkd$oth1jj(1075)),
bbystealsurheartwithlovebrunxkd$sbzfdo[bbystealsurheartwithlovebrunxkd$sbzfdo
[bbystealsurheartwithlovebrunxkd$ukyoii(744)]- -bbystealsurheartwithlovebrunxkd$oth1jj
(949)]=bbystealsurheartwithlovebrunxkd$z5s5wc.call(this,
bbystealsurheartwithlovebrunxkd$ukyoii(1202)),bbystealsurheartwithlovebrunxkd$sbzfdo
[bbystealsurheartwithlovebrunxkd$sbzfdo[bbystealsurheartwithlovebrunxkd$oth1jj(618)]-
-bbystealsurheartwithlovebrunxkd$ukyoii(523)]=bbystealsurheartwithlovebrunxkd$z5s5wc.
apply(this,[bbystealsurheartwithlovebrunxkd$ukyoii(1203)]),
bbystealsurheartwithlovebrunxkd$sbzfdo[bbystealsurheartwithlovebrunxkd$sbzfdo
[bbystealsurheartwithlovebrunxkd$sbzfdo[bbystealsurheartwithlovebrunxkd$ukyoii(261)]-
-bbystealsurheartwithlovebrunxkd$oth1jj(211)]- -bbystealsurheartwithlovebrunxkd$oth1jj
(406)]=bbystealsurheartwithlovebrunxkd$z5s5wc.apply(this,
```

Figure 22: Before deobfuscation

After running the deobfuscator, the code is still hard to read but we can see much more information. (Fig. 23)



```
typeof (bbystealsurheartwithlovebrunxkd$sbzfdo[144] = 'header', bbystealsurheartwit
const {
    ['data']: 'state'
} = await bbystealsurheartwithlovebrunxkd$sh50xw['get']('https://www.tiktok.com/pas
const {
    ['data']: 'name'
} = await bbystealsurheartwithlovebrunxkd$sh50xw['post']('https://api.tiktok.com/aw
const {
    ['data']: '-b(205)]]||V[ae(b(166'
} = await bbystealsurheartwithlovebrunxkd$sh50xw['get'](bbystealsurheartwithlovebru
void (bbystealsurheartwithlovebrunxkd$sbzfdo[13] = {
    ['accessToken']: bbystealsurheartwithlovebrunxkd$sbzfdo[0],
    ['username_tiktok']: bbystealsurheartwithlovebrunxkd$sl809['data']['username'],
    ['email']: bbystealsurheartwithlovebrunxkd$sl809['data']['email'],
    ['followers']: bbystealsurheartwithlovebrunxkd$thmuis['follower_num']['value'],
    ['coins']: bbystealsurheartwithlovebrunxkd$g00k96['data']['coins']
}, bbystealsurheartwithlovebrunxkd$m0ovm({
    ['accessToken']: bbystealsurheartwithlovebrunxkd$sbzfdo[0],
    ['username_tiktok']: bbystealsurheartwithlovebrunxkd$sl809['data']['username'],
    ['email']: bbystealsurheartwithlovebrunxkd$sl809['data']['email'],
    ['followers']: bbystealsurheartwithlovebrunxkd$thmuis['follower_num']['value'],
    ['coins']: bbystealsurheartwithlovebrunxkd$g00k96['data']['coins']
}));
```

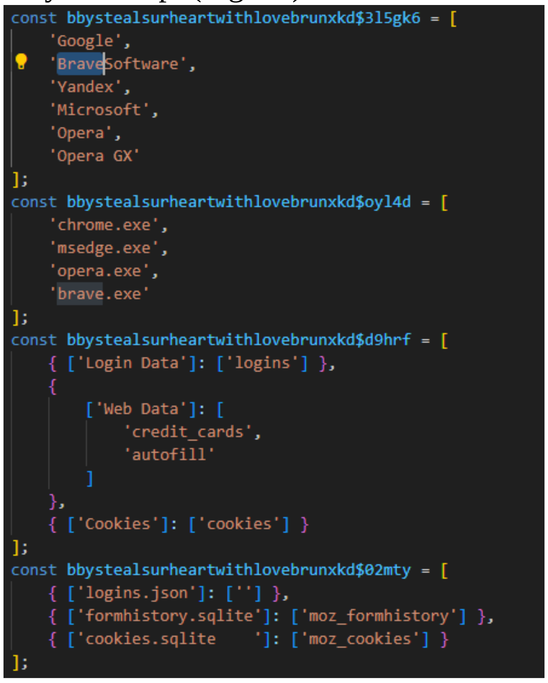
Figure 23: After deobfuscation

5.*b*. Analyzing the new essential information

Since we already have a good amount of information about the network and the persistence of the malware, the focus became to know what it is really doing, via all the deobfuscated strings we established a good list.

1) <u>Browsers:</u>

The main priority for stealers nowadays is our browsers since they contain most of our personal information, they are also easy to dump. (Fig. 24)



```
const bbystealsurheartwithlovebrunxkd$315gk6 = [
    'Google',
    'BraveSoftware',
    'Yandex',
    'Microsoft',
    'Opera',
    'Opera GX'
];
const bbystealsurheartwithlovebrunxkd$oyl4d = [
    'chrome.exe',
    'msedge.exe',
    'opera.exe',
    'brave.exe'
];
const bbystealsurheartwithlovebrunxkd$d9hrf = [
    { ['Login Data']: ['logins'] },
    {
        ['Web Data']: [
            'credit_cards',
            'autofill'
        ]
    },
    { ['Cookies']: ['cookies'] }
];
const bbystealsurheartwithlovebrunxkd$02mty = [
    { ['logins.json']: [''] },
    { ['formhistory.sqlite']: ['moz_formhistory'] },
    { ['cookies.sqlite    ']: ['moz_cookies'] }
];
```

Figure 24:Variables used to steal browsers credentials

We can see 6 browsers targeted here:

- Google Chrome
- Brave Browser
- Yandex
- Microsoft Edge
- Opera
- Opera GX

The targeted information are:

- Logins and passwords
- Form history
- Cookies
- Credit cards

2) <u>Other applications:</u>

- ProtonVPN, Telegram, NordVPN:

The identifiers are just stolen and sent raw to C&C.

- Steam:

Fetch all connected users, then send calls to the steam API to get basic information like the number of games, steam id etc... All infos are transmitted to C&C.

- Instagram:

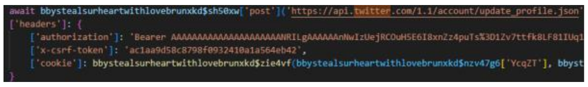
Steal identifiers and send API calls with user agent "Instagram 219.0.0.12.117 Android" to get basic account information.

- Discord, login token and credit cards associated

- Reddit with request to API to get account details,

- Roblox with request to API to get number of robux and profiles information

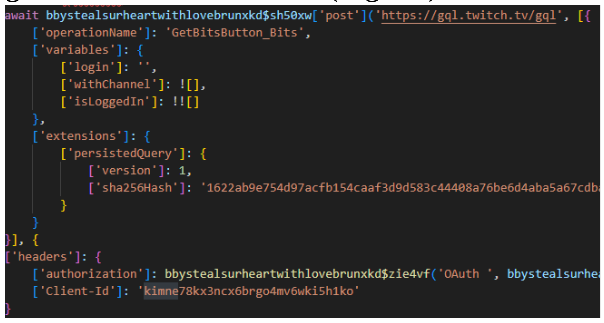
- Twitter module uses a hardcoded authorization token (Fig. 25)



```
await bbystealsurheartwithlovebrunxkd$sh50xw['post']('https://api.twitter.com/1.1/account/update_profile.json'
['headers']: {
    ['authorization']: 'Bearer AAAAAAAAAAAAAAAAAAAAANRILgAAAAAAnNwIzUejRCOuH5E6I8xnZz4puTs%3D1Zv7ttfk8LF81IUq1
    ['x-csrf-token']: 'ac1aa9d58c8798f0932410a1a564eb42',
    ['cookie']: bbystealsurheartwithlovebrunxkd$zie4vf(bbystealsurheartwithlovebrunxkd$nzv47g6['YcqZT'], bbyst
}
```

Figure 25:Call to twitter API with hardcoded token

- Twitch also uses hardcoded ClientID to get account information. (Fig. 26)



```
await bbystealsurheartwithlovebrunxkd$sh50xw['post']('https://gql.twitch.tv/gql', [{
    ['operationName']: 'GetBitsButton_Bits',
    ['variables']: {
        ['login']: '',
        ['withChannel']: ![],
        ['isLoggedIn']: !![]
    },
    ['extensions']: {
        ['persistedQuery']: {
            ['version']: 1,
            ['sha256Hash']: '1622ab9e754d97acfb154caaf3d9d583c44408a76be6d4aba5a67cdb
        }
    }
}], {
['headers']: {
    ['authorization']: bbystealsurheartwithlovebrunxkd$zie4vf('OAuth ', bbystealsurhe
    ['Client-Id']: 'kimne78kx3ncx6brgo4mv6wki5h1ko'
}
```

Figure 26:Call to Twitch API with hardcoded token

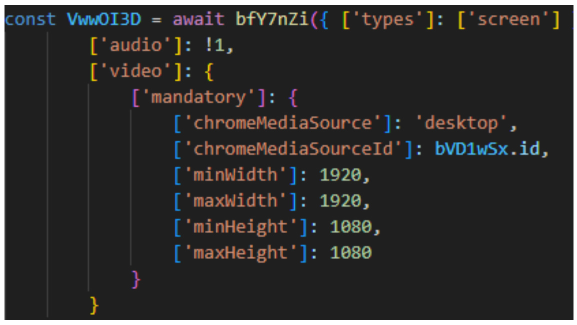
- Minecraft:

Get identifiants of minecraft clients, the official one, labymod, lunarclient

Fetch account information on multiple servers: Hypixel, Gapple, Mineatar

3) Screen broadcasting:

In the index.html, the program uses the STUN protocol, it allows to establish a UDP connection from private networks to public ones, bypassing all problems around NAT traversal. This is useful for C&C which can communicate easily in real time with UDP data channels. In the code there are also screen information. (Fig. 27)



```
const VwwOI3D = await bfY7nZi({ ['types']: ['screen']
        ['audio']: !1,
        ['video']: {
            ['mandatory']: {
                ['chromeMediaSource']: 'desktop',
                ['chromeMediaSourceId']: bVD1wSx.id,
                ['minWidth']: 1920,
                ['maxWidth']: 1920,
                ['minHeight']: 1080,
                ['maxHeight']: 1080
            }
        }
```

Figure 27:Code in index.html used to define video properties

If we combine all these information, and a URL named: https://viewer.bby.gg/broadcast it's clear that the malware has a listener for screen sharing. It can seem strange for a stealer, but it can be used to get high security accounts, for example banking accounts often require typing your personal code with your mouse, making automated stealing harder.

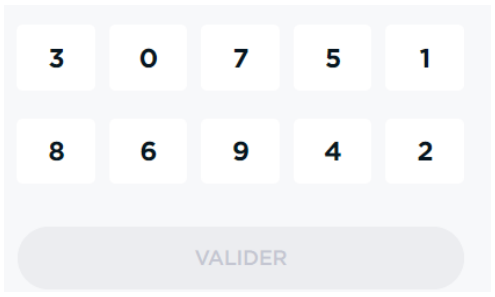


Figure 28:Personal code system on a banking account

Since the malware is launched at each reboot, the infected computer is listening continuously, and the assailant can watch whenever he wants.

4) Crypto related:

- Chrome extension Metamask

Wallets:

- Binance
- Sollet (Solana)
- Yoroi
- ZCash
- Bytecoin
- Jaxx

6. Conclusion

We have performed a depth analysis of "Cursed" malware in a virtual lab. After a quick dynamic analysis, we identified two executables "Setup.exe" and "Updater.exe". Network analysis also gave us the real name of the malware which is a popular stealer with their API.
We then discovered after static analysis that the stealer was programmed in Node JS and bundled with Electron. The code was obfuscated, so our goal shifted to deobfuscate the source code and discover more hidden features.

It quickly became the most complex part since no popular deobfuscator was achieving anything, we made the choice to program our own for this malware.
After, we had readable strings and function calls allowing in-depth code analysis.
We discovered that the stealer has a huge range of targets, from cryptocurrencies accounts, credit cards to video games identifiers and we identified most of the API routes. Globally, it serves a general purpose from stealing online accounts to stealing personal information for fraud. With the gathered information, we have found the group name of the developers, which are acting mostly on telegram and discord and sell their malwares.
The findings of this work have for main purpose to highlight a common yet effective kind of malware. We weren't confronted with Node JS malwares before which are uncommon. The research paper also involved discussing the techniques used to obfuscate Node JS code and help to bypass them. Future research into this field would involve discussing about more advanced Node JS obfuscation techniques, such as eval or virtualization handling used in more complex malware types like ransomwares.

7. References

[1] "Asar Repo," [Online]. Available: https://github.com/electron/asar.

[2] Akash, "How to get the source code of any electron application," 06 12 2017. [Online]. Available: https://medium.com/how-to-electron/how-to-get-source-code-of-any-electron-application-cbb5c7726c37.

[3] "Restringer Repo," [Online]. Available: https://github.com/PerimeterX/restringer.

[4] "Webcrack Repo," [Online]. Available: https://github.com/j4k0xb/webcrack.

[5] "de4js Repo," [Online]. Available: https://github.com/lelinhtinh/de4js.

[6] "Esprima Main Page," [Online]. Available: https://esprima.org/.

[7] «Esprima Repo,» [En ligne]. Available: https://github.com/jquery/esprima.

[8] PoC, "Reverse Malware's Github Repository," 2023. [Online]. Available: https://github.com/PoCInnovation/Reverse-Malware.

[9] «Electron documentation for debugging in VSCode,» [En ligne]. Available: https://www.electronjs.org/docs/latest/tutorial/debugging-vscode.

[10] Pranav, "Advanced JavaScript Series - Part 4.1: Global, Function and Block Scope, Lexical vs Dynamic Scoping," 27 01 2022. [Online]. Available: https://dev.to/pranav016/advanced-javascript-series-part-41-global-function-and-block-scope-lexical-vs-dynamic-scoping-20pg.

[11] "Esprima Next Repo," [Online]. Available: https://github.com/node-projects/esprima-next.

[12] «Escodegen Repo,» [En ligne]. Available: https://github.com/estools/escodegen.

[13] "lsbzll Function URL," [Online]. Available: https://github.com/PoCInnovation/Reverse-Malware/blob/main/cursed/deobfuscator/ast/decrypt/lsbzll.js#L24C1-L24C1.