../

Thu Dec 22 2022

authored by veritas

Reverse Engineering Tiktok's VM Obfuscation (Part 1)

TikTok has a reputation for its aggressive data collection. In fact, an article published on 22 December 2022 uncovered how ByteDance spied on multiple Forbes journalists using TikTok. While some of the data they collect may seem benign, it can be used to build a detailed profile of each user. Information such as user location, device type, and various hardware metrics are combined to create a unique "fingerprint" that can potentially be used to track a user's activity on and off the app. This data may also be used to prevent their APIs from being utilized in automated scripts by ensuring that the data from the requests seem humanlike.

The platform has implemented various methods to make it difficult for reverse-engineers to understand exactly what data is being collected and how it is being used. Analyzing the call stack of a request made on tiktok.com can begin to paint the picture for us. Let's start by doing a search for the term "food". Upon pressing enter, TikTok sends off a GET request with our search term and some extra telemetry embedded.

```
curl -G \
  -d 'aid=1988' \
  -d 'app_language=en' \
  -d 'app_name=tiktok_web' \
  -d 'battery_info=1' \
  -d 'browser_language=en-US' \
  -d 'browser_name=Mozilla' \
  -d 'browser_online=true' \
```

```
  -d 'browser_platform=Win32' \
  -d 'browser_version=5.0%20%28Windows%20NT%2010.0%3B%20Win64%3B%20x64%29%20Ap
  -d 'channel=tiktok_web' \
  -d 'cookie_enabled=true' \
  -d 'cursor=0' \
  -d 'device_id=7161571420764997166' \
  -d 'device_platform=web_pc' \
  -d 'focus_state=true' \
  -d 'from_page=search' \
  -d 'history_len=4' \
  -d 'is_fullscreen=false' \
  -d 'is_page_visible=true' \
  -d 'keyword=food' \
  -d 'os=windows' \
  -d 'priority_region=' \
  -d 'region=US' \
  -d 'screen_height=1440' \
  -d 'screen_width=2560' \
  -d 'tz_name=America%2FNew_York' \
  -d 'webcast_language=en' \
  -d 'msToken=e-Jl8_Qj4uCc5on6ZkVO2-NaZA8N4e6bNJbot-BuFM9HJI-9dA4zBMyaImxHWXwE
  -d 'X-Bogus=DFSzswVL3sTANG9HSkkrBGXyYJWI' \
  -d '_signature=_02B4Z6wo00001da8CfQAAIDA9R0nWiG.if3WvA1AABYN0b' \
'https://us.tiktok.com/api/search/user/full/' \
  -H 'authority: us.tiktok.com' \
  -H 'accept: */*' \
  -H 'accept-language: en-US,en;q=0.9' \
  -H 'origin: https://www.tiktok.com' \
  -H 'user-agent: Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36
--compressed
```

The response for this request is exactly what we'd expect: The JSON representation of accounts starting with or containing the keyword food.


```
{
  "type": 1,
  "user_list": [
```

```
{
  "user_info": {
    "uid": "6756983778017313798",
    "nickname": "Food Network",
    "signature": "The official Food Network TikTok!",
    "avatar_thumb": {
      "uri": "musically-maliva-obj/1661616615752710",
      "url_list": [
        "https://p16-sign-va.tiktokcdn.com/musically-maliva-obj/166161661
        "https://p16-sign-va.tiktokcdn.com/musically-maliva-obj/166161661
      ],
      "width": 720,
      "height": 720
    },
    "follow_status": 0,
    "follower_count": 3400000,
    "custom_verify": "",
    "unique_id": "foodnetwork"
  }
},
{
  "user_info": {
    "uid": "67676215429039401O2",
    "nickname": "Foodies",
    "signature": "FOODIES NATION \\nBiz - contact@ifoodies.co\\nMY LINKS ⬇
    "avatar_thumb": {
      "uri": "musically-maliva-obj/ae34b3144f24dd17f3810e2c04e41efd",
      "url_list": [
        "https://p16-sign-va.tiktokcdn.com/musically-maliva-obj/ae34b3144f
        "https://p16-sign-va.tiktokcdn.com/musically-maliva-obj/ae34b3144f
      ],
      "width": 720,
      "height": 720
    },
    "follow_status": 0,
    "follower_count": 21800000,
    "custom_verify": "Verified account",
    "unique_id": "foodies"
  }
```

```
    }
  ],
  "cursor": 10,
  "has_more": 1,
  "input_keyword": "food",
  "feedback_type": "user"
}
```


Most of the query parameters are self explanatory but there's three that stand out:

- msToken
- X-Bogus
- _signature

Removal of the **`_signature`** query parameter doesn't seem to have an affect as the request still goes through as expected but removal of any other parameter causes TikTok to give a 0 length response.

How are these parameters generated? Taking a look at the call stack tells us the journey from beginning to end.

```
Request call stack
  fetch          @ init.js?cache:1
  (anonymous)    @ browser-nocookie.lite.1.2.4.maliva.js:1
  window.fetch   @ secsdk-lastest.umd.js:5
  _0x290c10      @ webmssdk.js:1
  _0x30599d      @ webmssdk.js:1
  (anonymous)    @ webmssdk_ex.js:1
  (anonymous)    @ webmssdk_ex.js:1
  _0x3982a4      @ webmssdk_ex.js:1
  ...
```

The call to window.fetch being located in script **`secsdk-lastest.umd.js`** tells us that the fetch function has been monkey patched to provide additional functionality but perhaps what's more interesting are the obfuscated function names underneath.

An examination of the **`webmssdk.js`** script reveals that the code is intentionally made difficult to understand through obfuscation, as evidenced by the following function:

```
function _0x4e353d(_0x455c68, _0x1d9108) {
  var _0x10ee06 = parseInt(_0x455c68['slice'](_0x1d9108, _0x1d9108 + (-0x3e *
  return _0x10ee06 >>> 0x2354 + -0xd7d * 0x1 + -0x15d0 == 0x2432 + 0x2fc + -0x
  [0x246f * -0x1 + 0x11ac + 0x12c5, _0x10ee06 += parseInt(_0x455c68['slice'](_
  [0x9 * 0x41c + 0x7f * 0x2f + -0x1e25 * 0x2, _0x10ee06 += parseInt(_0x455c68[
}
```

View the fully obfuscated script over at [webmssdk.js]

By utilizing the [Babel suite], we are able to parse the source code and manipulate its Abstract Syntax Tree (AST). With this, we can create a simple transformation that reduces complex binary expressions to a single constant. The transformation code appears as follows:

```
import { ParseResult } from "@babel/parser";
import { File, numericLiteral } from '@babel/types';
import traverse from '@babel/traverse';

// 0x18e9 + 0x1 * 0x89c + -0x2185 * 0x1 -> 0
export function collapseBinaryExpressions(ast: ParseResult<File>) {
    traverse(ast, {
        BinaryExpression(path) {
            const { confident, value } = path.evaluate();
            if (!confident) return;

            path.replaceWith(numericLiteral(value));
        }
    })
}
```

The function, previously obfuscated, now appears in the following form:

```
function _0x4e353d(_0x455c68, _0x1d9108) {
 var _0x10ee06 = parseInt(_0x455c68['slice'](_0x1d9108, _0x1d9108 + 2), 16);
```

```
  return _0x10ee06 >>> 7 == 0 ? [1, _0x10ee06] : _0x10ee06 >>> 6 == 2 ? (_0x10e
}
```

Much better, but now we're stuck in ternary hell. We can create another simple transformation to unpack the nested ternary logic and make it more easily understood:

```
import { ParseResult } from "@babel/parser";
import { File, expressionStatement, blockStatement, returnStatement, ifStateme
import traverse from '@babel/traverse';

export function expandTernary(ast: ParseResult<File>) {
    traverse(ast, {
        ConditionalExpression(path) {
          const { consequent, alternate } = path.node;

          const consequentExpStatement = expressionStatement(consequent);
          const consequentBlock = blockStatement([returnStatement(consequentEx

          const alternateExpStatement = expressionStatement(alternate);
          const alternateBlock = blockStatement([returnStatement(alternateExpS

          path.parentPath.replaceWith(ifStatement(path.node.test, consequentBl
          path.skip();
        }
    })
}
```

Applying this transformation to function _0x4e353d produces the following result:

```
function _0x4e353d(_0x455c68, _0x1d9108) {
  var _0x10ee06 = parseInt(_0x455c68['slice'](_0x1d9108, _0x1d9108 + 2), 16);

  if (_0x10ee06 >>> 7 == 0) {
    return [1, _0x10ee06];
  } else {
    if (_0x10ee06 >>> 6 == 2) {
```

```
      return _0x10ee06 = (63 & _0x10ee06) << 8, [2, _0x10ee06 += parseInt(_0x4
    } else {
      return _0x10ee06 = (63 & _0x10ee06) << 16, [3, _0x10ee06 += parseInt(_0x
    }
  }
}
```

We could create more complex transformations to further improve the readability of the obfuscated script, but for the purposes of this article, these two transformations are sufficient.

As you review the script, you may notice recurring patterns. For example, consider these two function calls:

```
_0x8d6b0f('484e4f4a403f524300181220088de2ac00001280257e0df20000132e11020d12000
    0x0: decodeURIComponent,
    0x1: encodeURI,

    get 0x2() {
      return window;
    },

    get 0x3() {
      return URL;
    },

    get 0x4() {
      return setTimeout;
    },

    get 0x5() {
      return Request;
    },

    0x6: Object,

    get 0x7() {
      return Headers;
```

```
    },

    get 0x8() {
      return 'undefined' != typeof fetch ? fetch : void 0;
    },

    0x9: Array,
    0xa: JSON,

    get 0xb() {
      return _0x31cebc;
    },

    ...,

    0x17: RegExp
  }, void 0);

  _0x1ddb44('484e4f4a403f52430036112c7f259c75000001c45c5e31e2000001ea11000211000
    0x0: Math,

    get 0x1() {
      return window;
    },

    get 0x2() {
      return _0xad96ae;
    },

    set 0x2(_0x41c4f0) {
      _0xad96ae = _0x41c4f0;
    }
  }, void 0)
```

They follow a very similar schema: A function call with 3 parameters:

1. A string of alphanumeric characters that is not immediately recognizable as to its purpose.

2. An object containing getters and setters referencing various browser APIs and global variables.
3. void 0 (a fancy obfuscated way of saying undefined)

An exercise to you: Dump all function calls that meet the criteria listed above

To determine how this string is being used, we need to analyze the function it is being called in.

```
function _0x8d6b0f(weirdString, variablesObject, undef) {
  function _0x4e353d(_0x455c68, _0x1d9108) {
    var _0x10ee06 = parseInt(_0x455c68['slice'](_0x1d9108, _0x1d9108 + 2), 16)

    if (_0x10ee06 >>> 7 == 0) {
      return [1, _0x10ee06];
    } else {
      if (_0x10ee06 >>> 6 == 2) {
        _0x10ee06 = (63 & _0x10ee06) << 8;
        return [2, _0x10ee06 += parseInt(_0x455c68['slice'](_0x1d9108 + 2, _0x
      } else {
        _0x10ee06 = (63 & _0x10ee06) << 16;
        return [3, _0x10ee06 += parseInt(_0x455c68['slice'](_0x1d9108 + 2, _0x
      }
    }
  }

  var _0x297812,
      _0x4fab32 = 0,
      _0x1e4e7c = [],
      _0x272a95 = [],
      _0x189f25 = parseInt(weirdString['slice'](0, 8), 16),
      _0x448290 = parseInt(weirdString['slice'](8, 16), 16);

  if (1213091658 !== _0x189f25 || 1077891651 !== _0x448290) throw new Error('m
  if (0 !== parseInt(weirdString['slice'](16, 18), 16)) throw new Error('ve');

  for (_0x297812 = 0; _0x297812 < 4; ++_0x297812)
```

```
    _0x4fab32 += (3 & parseInt(weirdString['slice'](24 + 2 * _0x297812, 26 + 2

  var _0x1ee8ce = parseInt(weirdString['slice'](32, 40), 16);
  var _0x330964 = 2 * parseInt(weirdString['slice'](48, 56), 16);

  for (_0x297812 = 56; _0x297812 < _0x330964 + 56; _0x297812 += 2)
    _0x1e4e7c['push'](parseInt(weirdString['slice'](_0x297812, _0x297812 + 2),

  var _0x5541d2 = _0x330964 + 56,
      _0x5dd331 = parseInt(weirdString['slice'](_0x5541d2, _0x5541d2 + 4), 16)

  for (_0x5541d2 += 4, _0x297812 = 0; _0x297812 < _0x5dd331; ++_0x297812) {
    var _0x2e1d65 = _0x4e353d(weirdString, _0x5541d2);

    _0x5541d2 += 2 * _0x2e1d65[0];

    for (var _0x2a7c5a = '', _0x591b13 = 0; _0x591b13 < _0x2e1d65[1]; ++_0x591
      var _0x301a0f = _0x4e353d(weirdString, _0x5541d2);

      _0x2a7c5a += String['fromCharCode'](_0x4fab32 ^ _0x301a0f[1]), _0x5541d2
    }

    _0x272a95['push'](_0x2a7c5a);
  }

  // further code omitted ...
}
```

We can immediately see that the function we deobfuscated earlier is defined within the **`_0x8d6b0f`** function. Additionally, the argument names have been made more readable for ease of understanding.

```
_0x189f25 = parseInt(weirdString['slice'](0, 8), 16),
_0x448290 = parseInt(weirdString['slice'](8, 16), 16);

if (1213091658 !== _0x189f25 || 1077891651 !== _0x448290) throw new Error('mhe
if (0 !== parseInt(weirdString['slice'](16, 18), 16)) throw new Error('ve');
```

The first 16 characters are evenly split into two parts and then converted into an integer from base 16. The result is then compared to two magic constants: **`1213091658`** and **`1077891651`**. Applying this logic to our string will result in it passing these checks.

```
parseInt("484e4f4a", 16); // 1213091658
parseInt("403f5243", 16); // 1077891651
```

A check for a 00 separator follows immediately after. While we are still unsure of the exact purpose of this string, we have determined how it should start.

```
for (_0x297812 = 0; _0x297812 < 4; ++_0x297812)
  _0x4fab32 += (3 & parseInt(weirdString['slice'](24 + 2 * _0x297812, 26 + 2 *
```

Characters 24-34 are divided into parts and used in some bitwise arithmetic that is calculated for the variable **`_0x4fab32`**. Searching for the use of this variable leads us to a call to the **`String#fromCharCode`** function, where it is XORed with another variable.

```
for (var _0x2a7c5a = '', _0x591b13 = 0; _0x591b13 < _0x2e1d65[1]; ++_0x591b13)
  var _0x301a0f = _0x4e353d(weirdString, _0x5541d2);

  _0x2a7c5a += String['fromCharCode'](_0x4fab32 ^ _0x301a0f[1]);
  _0x5541d2 += 2 * _0x301a0f[0];
}
```

This strongly suggests the use of an [XOR cipher], leading me to conclude that variable **`_0x4fab32`** is likely the key. Based on this discovery, we can infer the purpose of nearby variables. The decryption process now looks as follows:

```
var _0x330964 = 2 * parseInt(weirdString['slice'](48, 56), 16);

var instructionPointer = _0x330964 + 56; // where the strings section starts
var amountOfStrings = parseInt(weirdString['slice'](instructionPointer, instru
```

```
for (instructionPointer += 4, curIdx = 0; curIdx < amountOfStrings; ++curIdx)
  // first item in the array is the length of the opcode
  // second item is the length of the string
  var opcodeLenAndStringLen = readOpcode(weirdString, instructionPointer);

  instructionPointer += 2 * opcodeLenAndStringLen[0];

  for (var decryptedStr = '', curCharIdx = 0; curCharIdx < opcodeLenAndStringL
    // first item in the array is the length of the opcode
    // the second item is the character that is encrypted
    var opcodeLenAndCharacter = readOpcode(weirdString, instructionPointer);

    decryptedStr += String['fromCharCode'](key ^ opcodeLenAndCharacter[1]);
    instructionPointer += 2 * opcodeLenAndCharacter[0];
  }

  decryptedStrings['push'](decryptedStr);
}
```

With all the necessary pieces in place, we can now isolate this logic and potentially retrieve strings from the previously mentioned long and obfuscated string. I chose to implement this in TypeScript and run it using Node, but the logic can be implemented in any language of your choosing.

```
const MAGIC_1 = 1213091658;
const MAGIC_2 = 1077891651;

function toBase10(base16Str: string) {
  return parseInt(base16Str, 16);
}

function buildKey(bytecode: string) {
  let key = 0;
  for (let i = 0; i < 4; i++) {
    key += (3 & toBase10(bytecode.slice(24 + 2 * i, 26 + 2 * i))) << 2 * i;
  }
  return key;
```

```
}

function readOpcode(bytecode: string, instructionPointer: number) {
  var opcode = toBase10(bytecode.slice(instructionPointer, instructionPointer
  if (opcode >>> 7 == 0) {
    return [1, opcode];
  } else if (opcode >>> 6 == 2) {
    opcode = (63 & opcode) << 8
    return [2, opcode += toBase10(bytecode.slice(instructionPointer + 2, instr
  } else {
    opcode = (63 & opcode) << 16;
    return [3, opcode += toBase10(bytecode.slice(instructionPointer + 2, instr
  }
}

function getStringsDecoded(bytecode: string, stringDataLocation: number, strir
  let instructionPointer = stringDataLocation;
  const strings: string[] = [];
  for (let i = 0; i < stringCount; ++i) {
    const [opcodeLength, stringLength] = readOpcode(bytecode, instructionPoint
    instructionPointer += 2 * opcodeLength;
    let stringBuffer = '';
    for(let curCharIdx = 0; curCharIdx < stringLength; ++curCharIdx) {
      const [opcodeLength, encryptedChar] = readOpcode(bytecode, instructionPo
      stringBuffer += String.fromCharCode(key ^ encryptedChar);
      instructionPointer += 2 * opcodeLength;
    }
    strings.push(stringBuffer);
  }
  return strings;
}

function run(bytecode: string) {
  const magicValue1 = toBase10(bytecode.slice(0, 8));
  const magicValue2 = toBase10(bytecode.slice(8, 16));

  if (magicValue1 != MAGIC_1 || magicValue2 != MAGIC_2)
    throw new Error("bad bytecode: magic values not found");
```

```
  if (toBase10(bytecode.slice(16, 18)) !== 0)
    throw new Error("bad bytecode: no separator found after magic values");

  const key = buildKey(bytecode);
  const instructionPointer = 2 * toBase10(bytecode.slice(48, 56));
  const stringDataLocation = instructionPointer + 56;
  const stringCount = toBase10(bytecode.slice(stringDataLocation, stringDataLc
  const strings = getStringsDecoded(bytecode, stringDataLocation + 4, stringCc
  console.log(strings);
}

const bytecode = process.argv[2];
run(bytecode);
```

If we run our script using the initial bytecode:

```
$ ts-node src/vm.ts 484e4f4a403f52430036112c7f259c75000001c45c5e31e2000001ea11
```

We obtain the following output:

```
[
  'regionConf',
  'host',
  'indexOf',
  'sec',
  'asgw',
  '?',
  'substr',
  'split',
  "'",
  'join',
  '%27',
  '',
  'length',
  '&',
  '=',
  '_mssdk',
  '_enablePathListRegex',
```

```
'test',
'application/x-www-form-urlencoded',
'application/json',
'X-Bogus',
'_signature',
'XMLHttpRequest',
'prototype',
'open',
'setRequestHeader',
'send',
'overrideMimeType',
'_ac_intercepted',
'_send',
'_byted_intercept_list',
'push',
'func',
'arguments',
'^content-type$',
'i',
'toString',
'toLowerCase',
';',
'_byted_content',
'apply',
'_overrideMimeTypeArgs',
'toUpperCase',
'_byted_method',
'_byted_url',
'onabort',
'onerror',
'onload',
'onloadend',
'onloadstart',
'onprogress',
'ontimeout',
'GET',
'POST',
'_signature=',
'_byted_body',
```

```
'onreadystatechange',
'upload',
'msStatus',
'__ac_testid',
'msToken',
'v',
'url',
'forreal',
'sdi',
'secInfoHeader',
'responseURL',
'href',
'location',
'getResponseHeader',
'x-ms-token',
'msNewTokenList',
'slardarErrs',
'init',
'function',
'__ac_intercepted_fetch',
'fetch',
'_fetch',
'then',
'ok',
'headers',
'get',
'method',
'set',
'clone',
'text',
'body',
'referrer',
'referrerPolicy',
'mode',
'credentials',
'cache',
'redirect',
'integrity',
'content-type',
```

```
  'keys',
  'bodyVal2str',
  'parse',
  '_urlRewriteRules',
  'replace',
  ... 2 more items
]
```

Great, we were able to successfully extract all strings from this particular module. We even see the strings **`_signature`** and **`X-Bogus`**! If we run our script using the strange string from the second function, we obtain a completely separate set of strings.

```
ts-node src/vm.ts 484e4f4a403f52430036112c7f259c75000001c45c5e31e2000001ea1100
```

```
[
  'floor',       'matchMedia',
  '(',           'concat',
  ': ',          ')',
  'matches',     'length',
  'function',    'resolution',
  '1.5',         'dppx',
  'orientation', 'landscape',
  'portrait',    '',
  'hover',       'none',
  'any-pointer', 'coarse',
  'fine',        'anyPointer',
  'max-height',  'px',
  'maxHeight',   'max-width',
  'maxWidth',    'max-resolution',
  'dpi'
]
```

This is because each "weird string" is actually bytecode that is interpreted and executed by TikTok's custom virtual machine to perform various tasks. Many modules handle bot protection and fingerprinting in their own ways.

For instance, this module is responsible for managing canvas fingerprinting, in which a user's machine's rendering of an HTML5 canvas element is used to create a fingerprint for them:

```
484e4f4a403f5243000c2d350434d3a0000005d7531c1682000005f3110001022334000511000
[
  'object',
  'length',
  '🐼OynG@%tp$',
  'rgba(47, 211, 69, .99)',
  '*+(}#?🐼 🎅',
  'rgba(150, 32, 170, .97)',
  'rgba(255, 12, 220, 1)',
  '',
  'createElement',
  'canvas',
  'width',
  'height',
  'getContext',
  '2d',
  'createLinearGradient',
  'addColorStop',
  'red',
  '0.1',
  'white',
  '0.2',
  'blue',
  '0.3',
  'yellow',
  '0.4',
  'purple',
  '0.7',
  'orange',
  'magenta',
  'fillStyle',
  'fillRect',
  'green',
  '0.5',
```

```
  'beginPath',
  'arc',
  'PI',
  'stroke',
  '12px Sans',
  'font',
  'top',
  'textBaseline',
  'fillText',
  'shadowBlur',
  'showOffsetX',
  '14px Sans',
  'strokeStyle',
  'toDataURL',
  'image/png',
  'getImageData',
  'a',
  'b',
  'c',
  'toString',
  'd',
  'e'
]
```

Here are the strings for TikTok's WebGL module, which can be used to gather your vendor and other GPU information:

```
484e4f4a403f5243001f273cf4f0727900000366f96a1ef700000386110000120000030029330
[
  'length',
  '',
  'WEBGL',
  'VENDOR',
  'RENDERER',
  '/',
  'webglData',
  'gpu',
  'getSupportedExtensions',
```

```
'supportedExtensions',
'getContextAttributes',
'antialias',
'getParameter',
'BLUE_BITS',
'blueBits',
'DEPTH_BITS',
'depthBits',
'GREEN_BITS',
'greenBits',
'maxAnisotropy',
'MAX_COMBINED_TEXTURE_IMAGE_UNITS',
'maxCombinedTextureImageUnits',
'MAX_CUBE_MAP_TEXTURE_SIZE',
'maxCubeMapTextureSize',
'MAX_FRAGMENT_UNIFORM_VECTORS',
'maxFragmentUniformVectors',
'MAX_RENDERBUFFER_SIZE',
'maxRenderbufferSize',
'MAX_TEXTURE_IMAGE_UNITS',
'maxTextureImageUnits',
'MAX_TEXTURE_SIZE',
'maxTextureSize',
'MAX_VARYING_VECTORS',
'maxVaryingVectors',
'MAX_VERTEX_ATTRIBS',
'maxVertexAttribs',
'MAX_VERTEX_TEXTURE_IMAGE_UNITS',
'maxVertexTextureImageUnits',
'MAX_VERTEX_UNIFORM_VECTORS',
'maxVertexUniformVectors',
'SHADING_LANGUAGE_VERSION',
'shadingLanguageVersion',
'STENCIL_BITS',
'stencilBits',
'VERSION',
'version',
'getExtension',
'WEBGL_debug_renderer_info',
```

```
  'UNMASKED_VENDOR_WEBGL',
  'UNMASKED_RENDERER_WEBGL',
  'assign',
  'vendor',
  'renderer',
  'createElement',
  'canvas',
  'getContext',
  'webgl',
  'experimental-webgl',
  'EXT_texture_filter_anisotropic',
  'WEBKIT_EXT_texture_filter_anisotropic',
  'MOZ_EXT_texture_filter_anisotropic',
  'MAX_TEXTURE_MAX_ANISOTROPY_EXT'
]
```

This article does not delve into the specifics of how these strings are utilized or how TikTok interprets the rest of the bytecode through its custom virtual machine and various opcodes. If that is something you are interested in, keep an eye out for the second part of this series :)

If you're interested in a full strings dump check out **strings.txt**

Mastodon
Twitter
Discord: veritas#0001
Email: f @ nullpt.rs