Exploring ScamClub Payloads via Deobfuscation Using Abstract Syntax Trees

blog.confiant.com/exploring-scamclub-payloads-via-deobfuscation-using-abstract-syntax-trees-65ef7f412537

BOZOSLIVEHERE September 28, 2023

Introduction

ScamClub is a prolific threat actor in the programmatic ad space known to carry out large-scale attacks with the purpose of scamming and defrauding their victims. ScamClub utilizes real-time bidding (RTB) integration with ad exchanges to push malicious JavaScript payloads upstream to their potential victims. These payloads attempt to forcefully redirect victims to any number of fraudulent pages such as phishing pages, gift card scams, giveaway scams, and more. More information about the ScamClub threat landscape and their modus operandi can be found in the ScamClub threat report.

In this article, we'll go over the de-obfuscation of the short version of the ScamClub stage two payload.

Payload Analysis

The ScamClub payloads come in three stages. The first stage is the creative, which is only lightly obfuscated and leads to the second stage — a payload that does the fingerprinting of potential victims to determine whether or not to continue with the forced redirects. The second stage of the ScamClub payloads comes in two distinct versions: short and long. The longer version of the payload contains everything present in the short version with some additional fingerprinting techniques. Interestingly, the long version of the payload has certain fingerprinting functions implemented but never called. In addition, the fingerprinting functions present in the short version are expanded upon to improve detection of security products or other evidence that the payload is being analyzed by an adversary.

The third stage of the ScamClub attack is the payload which performs the forced redirect attacks. A more in-depth analysis of the deobfuscated payload can be found in the ScamClub threat report and will be covered in even more depth in a future blog.

Obfuscation

All stages and versions of the ScamClub payload are obfuscated by the attackers using the same obfuscator. The obfuscator used by ScamClub is not available on the open internet, but has been observed to be used by various Chinese threat actors. The obfuscator uses two different layers of obfuscation. The first layer simply contains the second layer encoded with some basic encoding/encryption and is not heavily obfuscated per se. Once the first layer is decoded, we are presented with the second layer that uses some interesting obfuscation tricks to make the analysis of the sample more difficult. It should be noted that the first layer is encoded randomly, containing different variable names, function names, and encryption primitives in each sample. This makes signature-based detection more difficult completely eliminates the possibility of hash-based detection.

In order to completely deobfuscate ScamClub payloads, we are going to be writing a deobfuscator using the Babel library. Babel is a transpiler for JavaScript. A transpiler is a compiler that takes as input the source code of one language and produces as its output functionally-equivalent source code for the same language or another. Babel includes a parser that produces Abstract Syntax Trees (ASTs) for the source code it parses. We can use these ASTs to look for obfuscation techniques in the structure of the source code and convert them to deobfuscated, highly-readable source code. First, however, we need to unwrap the first layer of encoding to get to the obfuscation tricks used by the next layer to make the resulting decoded source code unreadable.

Obfuscation Layer 1 — Random Encoding

The payload's first layer of obfuscation looks like the following. Note that the encryption constants, variable names, and function names are different every time the payload is generated.

First, the payload sets up a function **rmH** that is used to decrypt the two long, encrypted strings seen later in the payload. These strings, once decrypted, contain the JavaScript code that is used to execute the rest of the payload. The first string looks like the following snippet when decrypted:

```
var n = 14,    b = 15,    t = 62;var x = "abcdefghijklmnopqrstuvwxyz";var k = [89,
75, 82, 76, 80, 94, 70, 71, 90, 86, 72, 60, 66, 85, 88, 74, 65, 79, 87, 81];var a =
[];for (var u = 0; u < k.length; u++) a[k[u]] = u + 1;var l = [];n += 19;b += 78;t +=
34;for (var j = 0; j < arguments.length; j++) {    var o = arguments[j].split(" ");
for (var z = o.length - 1; z >= 0; z--) {        var r = null;        var v = o[z];
var f = null;        var d = 0;        var c = v.length;        var y;        for
(var p = 0; p < c; p++) {            var m = v.charCodeAt(p);            var h =
a[m];            if (h) {                r = (h - 1) * b + v.charCodeAt(p + 1) - n;
y = p;                p++;            } else if (m == t) {                r = b *
(k.length - n + v.charCodeAt(p + 1)) + v.charCodeAt(p + 2) - n;                y = p;
p += 2;            } else {                continue;            }            if (f ==
null) f = [];            if (y > d) f.push(v.substring(d, y));            f.push(o[r
+ 1]);            d = p + 1;        }        if (f != null) {            if (d < c)
f.push(v.substring(d));            o[z] = f.join("");        }    }
l.push(o[0]);}var g = l.join("");var e = [92, 39, 10, 96, 32, 42].concat(k);var w =
String.fromCharCode(46);for (var u = 0; u < e.length; u++)    g = g.split(w +
x.charAt(u)).join(String.fromCharCode(e[u]));return g.split(w + "!").join(w);
```

As you can see, it is simply another decryption function used to decrypt the second long string from the payload. Once the second long string is decrypted, we end up with the main script obfuscated with a second layer of anti-analysis techniques.

Let's modify the original script to output the final decrypted function without executing it. This can be executed in your browser's console or directly with Node.js:

```
(function() {    // Truncated for brevity    var RGe = ArR(nYr, rmH(tlB));    var gik
= RGe(rmH('Nn}K0W6aY4)-{b(K%[...'));    console.log(beautify(gik, {indent_size:
2}));})();
```

Here are the results:

Obfuscation Layer Two— Obfuscated Main Script

Running the modified script above gives us the preceding output, which includes several interesting obfuscation tricks designed to make the analysis of the payloads a difficult and time-consuming process.

Now that we've reached the second layer of obfuscation, we see some new obfuscation techniques that are not only encoding, but tricks designed to make the analysis of the script more tedious. Let's go over them one by one.

String Encryption

All strings in the obfuscated payload are encrypted. Once the payload is run, all of the strings are decrypted at once and put into an array and any time a string is needed, the array is referenced. This makes the reading of the code extremely difficult as any time a string is needed one must reference the array of strings instead of viewing them directly in place.

```
var O = (bh)("tangMa9...", 865159);



function bh(h, j) {
  // string decryption code
}

function P(a) {  var f = {},    d = {};  f._ = a;  cg(f);  var c = bO()[O[24]](O[23])
[O[22]](0);  d._ = bO()[O[26]](O[25]);;  ch(d);  ci(d, f);  c[O[30]](d._)}
```

As you can see, all of the strings are decrypted when the payload is run and later, in the function **P()** they are referenced with the array they are stored in, **O**. Once the strings are decrypted and placed in-line, **P()** should look like this:

```
function P(a) {  var f = {},    d = {};  f._ = a;  cg(f);  var c = bO()
["getElementsByTagName"]("head")["item"](0);  d._ = bO()["createElement"]("script");
;  ch(d);  ci(d, f);  c["appendChild"](d._);}
```

Functions Returning Identifiers

In the function **P()** above, we can see that a few other functions are called: **bO()**, **cg()**, **ch()**, and **ci()**. Let's take a look at the function **bO()**
first:

```
function bO() {  return document}
```

There are many functions inside of the obfuscated payload that simply return operators, such as this one. In our deobfuscated code, we need to see those identifiers being referenced directly to make the code more readable. One way is to rename the function to something that makes sense with something like VSCode or any editor with LSP support, but we will be doing this correctly later with Babel. Once we apply our deobfuscation of identifiers wrapped in functions to the script, **P()** will look even better:

```
function P(a) {  var f = {},    d = {};  f._ = a;  cg(f);  var c =
document["getElementsByTagName"]("head")["item"](0);  d._ = document["createElement"]
("script");  ;  ch(d);  ci(d, f);  c["appendChild"](d._);}
```

Binary and Unary Operators in Functions

There are lots of different functions in the obfuscated code that wrap binary and unary operators with functions, making the code much more difficult to read. Let's take a look at another function, =bd()=, and the functions it calls:

```
function bd(d, f) {
  var a = bq((bo(d, 0xFFFF)), (bo(f, 0xFFFF)));
  var c = bq((bx(d, 16)) + (bx(f, 16)), (bx(a, 16)));
  return bk((bu(c, 16)), (bo(a, 0xFFFF)))
}



function bq(a, c) {
  return a + c
}



function bo(a, c) {
  return a & c
}



function bx(a, c) {
  return a >> c
}



function bk(a, c) {
  return a | c
}

function bu(a, c) {  return a << c}
```

As you can see, the functions **bq()**, **bo()**, **bx()**, **bk()**, and **bu()** all wrap simple operators in functions. The function **bd()** sure would look better and be easier to read with those operators in-line, as such:

```
function bd(d, f) {  var a = (d & 0xFFFF) + (f & 0xFFFF);  var c = ((d >> 16) + (f >>
16)) + (a >> 16);  return (c << 16) | (a & 0xFFFF);}
```

Now that we've taken a look at all of the different obfuscation techniques used, let's start figuring out how to remove them automatically one by one using Babel.

Babel

Babel is a transcompiler for JavaScript that includes a parser and transformer that we can use for our deobfuscator. Babel uses the Visitor Pattern, a software design pattern in which you create visitors, functions that are run against all elements of an Abstract Syntax Tree (AST). Using the visitor pattern, we go down the source tree, visiting every node, and apply our logic to that node to change the obfuscated code into something more readable.

In addition to a parser, Babel also contains a generator. This will allow us to generate code to replace the obfuscated code with something easier to analyze as well as generate code from our ASTs to run in the Node.js VM.

Abstract Syntax Trees

The Babel library can be used to parse our source code and produce Abstract Syntax Trees. These ASTs will be useful in defining the structure of the different techniques used by the obfuscation tool employed by ScamClub. Let's take a look at the very first function and generate an AST for it:

```
function bO() {  return document}
```

This function simply returns another identifier and nothing more. Let's generate an AST for it using the AST Explorer web tool:

```
{  "type": "FunctionDeclaration",  "id": {    "type": "Identifier",    "name": "bO"
},  "params": [],  "body": {    "type": "BlockStatement",    "body": [      {
"type": "ReturnStatement",        "argument": {          "type": "Identifier",
"name": "document"        }      }    ]  }}
```

Note that the output from the AST Explorer tool has been simplified — the tool returns much more information than listed above, but for the sake of brevity and clarity we have removed information such as line and column numbers. If you would like to see the whole output, see the results from the AST Explorer tool.

The AST returned includes information about the name of the function, the type of statement it includes, the contents of that statement — in this case, a return statement — and the identifier returned. In order to write our deobfuscator, we will need to define generic ASTs for all of the different obfuscation techniques used.

In the obfuscated payloads, there are many different functions that simply return an identifier — whether that identifier is a keyword such as **window** or **console** or a variable defined by the script itself. In order to make this AST generic, we want something like this:

```
{  "type": "FunctionDeclaration",  "id": {    "type": "Identifier",    "name": String
// Any string  },  "params": [],                    // length 0  "body": {
"type": "BlockStatement",    "body": [      {        "type": "ReturnStatement",
"argument": {          "type": "Identifier",          "name": String            //
Any string        }      }    ]  }}
```

We will later apply logic that matches this generic AST to transform the obfuscated code into something more readable using Babel.

Writing our Deobfuscator — Import Modules & Setup

First, we need to set up the modules we'll be using. We will use **babel/parser**, **babel/traverse**, **babel/generator**, **js-beautify**, and the built-ins **vm** and **fs**. In addition we will use **commander** to parse our command-line options.

```
const parser = require("@babel/parser");
const traverse = require("@babel/traverse").default;
const t = require("@babel/types");
const generate = require("@babel/generator").default;



const beautify = require("js-beautify");



const { readFileSync, writeFile } = require("fs");
const vm = require("vm");



const { program } = require('commander');

program  .option('-f, --file <file>');program.parse();const options = program.opts();
```

Next we need to configure the parser. In some ScamClub samples, the **return** keyword is used outside of a function, which causes Babel to break and complain about the incorrect use of **return**. To fix this, we need to enable the **allowReturnOutsideFunction** option:

```
const parserOptions = {  plugins: ['@babel/plugin-syntax-jsx'],
allowReturnOutsideFunction: true,};
```

Let's parse the code into an AST and set up a context for the VM we will use to execute parts of ScamClub's code in order to decrypt layer 1 of the obfuscation as well as the strings in layer 2:

```
let code = readFileSync(options.file, "utf8")let ast = parser.parse(code,
parserOptions);const decryptFuncCtx = vm.createContext();
```

Deobfuscation — Layer One

The first thing we need to do is identify the variable which receives the second layer once decrypted, so that we can execute the variable declaration in the VM and store the results to be able to continue deobfuscation. If we go back to the fully obfuscated code, we see the following line which decrypts this layer of obfuscation so that the script can continue to execute the second layer of obfuscated code.

```
var gik = RGe(rmH('...'));
```

Taking a look at the rest of the code, we need to try to see what makes this line unique from all others so that we can use a specific pattern to reliably match the AST and identify this line in all versions of ScamClub payloads. This line is the only line which matches the pattern **VariableDeclarationSomeVar=SomeFunc1(SomeFunc2(StringLiteral))**. Now, let's take a look at the AST of this line:

```
{    "body": [        {        "type": "VariableDeclaration",        "declarations":
[            {            "type": "VariableDeclarator",            "id": {
"type": "Identifier",                "identifierName": "gik",                "name":
"gik"            },            "init": {                "type": "CallExpression",
"callee": {                    "type": "Identifier",
"identifierName": "RGe",                    "name": "RGe"                },
"arguments": [                {                    "type": "CallExpression",
"callee": {                        "type": "Identifier",
"identifierName": "rmH",                        "name": "rmH"                    },
"arguments": [                    {                        "type": "StringLiteral"
}                    ]                }                ]            }            }
],        "kind": "var"        }    ]}
```

Now we're going to create our first visitor. This visitor should look for a **VariableDeclaration** type node whose **init** type is **CallExpression**. The first argument's type should also be **CallExpression**.

```
let stage2DecodedVarName = null;const variableInitIdentifierVisitor = {
VariableDeclaration(path) {    const init =
path.node.declarations[path.node.declarations.length - 1].init;    if(init &&
init.type === "CallExpression") {      const callee = init.arguments[0].callee;
const argType = init.arguments[0].type      if (argType === "CallExpression") {
stage2DecodedVarName = path.node.declarations[0].id.name;      }    }
}};traverse(ast, variableInitIdentifierVisitor); // Run visitor against AST
```

This visitor will traverse the whole script, stopping at each **VariableDeclaration** type node to check if the AST matches the pattern we're looking for. Once the pattern is found, we store the name of the variable being declared into **stage2DecodedVarName**. In the case of the script we're working on, the variable name is **gik**.

Using the VM to Decrypt Layer Two

Now we need run the code relevant to the decryption of the next layer in our VM context. In order to do this, we will want to run all **VariableDeclaration** nodes and the **FunctionDeclaration** nodes relevant to decryption.

```
let layer2Decoded = null
const decryptLayer2Visitor = {
  // We need to execute all VariableDeclaration nodes into our context
  VariableDeclaration(path) { // Run all variable declarations in the decrypt context
    const varDecCode = generate(path.node).code; // Generate the code to execute in
context
    vm.runInContext(varDecCode, decryptFuncCtx); // Execute the decryption function
delcaration in VM context
    if (path.node.declarations[0].id.name == stage2DecodedVarName) {
      decodeLayer2Code = generate(path.node.declarations[0].id).code;
      layer2Decoded = vm.runInContext(decodeLayer2Code, decryptFuncCtx)
      path.stop(); // Stop execution fo the visitor now that we've received the
information we're looking for
    }
  },



  // We also need the FunctionDeclaration node to be executed in our context to be
used
  FunctionDeclaration(path) {
    const funcDecCode = generate(path.node).code; // Generate the code to execute in
context
    vm.runInContext(funcDecCode, decryptFuncCtx); // Execute the decryption function
delcaration in VM context
    path.remove() // Remove the decryption function since it has served its use
  },
};

traverse(ast, decryptLayer2Visitor);
```

By running the **VariableDeclaration** and **FunctionDeclaration** nodes in our VM, we now have the variables and functions available to be executed within our VM context. In the case that the name of the variable being declared in the current node being visited matches **stage2DecodedVarName**, we generate some code to return the value of the variable after it's been declared. This is our decoded second layer. Now let's get this layer ready for further analysis:

```
// parse our newly decoded layer 2 into its ASTlet ast_layer2 =
parser.parse(layer2Decoded)// set up a new VM context for layer 2const layer2Ctx =
vm.createContext()// Beautify codelayer2Decoded = beautify(layer2Decoded, {
indent_size: 2,  space_in_empty_paren: true,});
```

Layer Two

Now we can start taking a look at layer two of the obfuscation. This is where the actual script itself executes and does its fingerprinting and insertion of a script element containing the forced redirect script. A number of different obfuscation techniques are used in the second layer, and we'll go over making visitors to deobfuscate them one by one.

Decrypting & Replacing Encrypted Strings In-Line

Let's go back and take a look at the String Encryption section. Let's make an AST out of the string decryption line and take a look at its structure so that we can build a visitor to decrypt the strings:

```
var O = (bh)("tangMa9...", 865159);

  {    "type": "VariableDeclaration",    "declarations": [      {        "type":
"VariableDeclarator",        "id": {          "type": "Identifier",          "name":
"O"        },        "init": {          "type": "CallExpression",          "callee":
{            "type": "Identifier",            "name": "bh",          },
"arguments": [            {              "type": "StringLiteral",
"extra": {               "rawValue": "tangMa9...",               "raw":
"\"tangMa9...\""              },              "value": "tangMa9..."            },
{              "type": "NumericLiteral",              "value": 865159            }
]        }      }    ],    "kind": "var"  }
```

Taking a look at this AST and the rest of the code, note that this line is the only line which matches an AST like **VariableDeclaration Identifier =CallExpression(StringLiteral, NumericLiteral)**. Let's make a visitor that looks for this pattern and then executes the string decryption function inside of the VM in order to give us the list of encrypted strings. Later, we'll replace the references to **O** (the strings array in the sample we're working with here) in-line to make the code more readable. We'll also need to first set up the VM context for layer two, so we'll create a short visitor to run all **FunctionDeclaration** nodes inside of our VM context. Running these nodes imports them into the context without actually executing them, allowing us to execute them later on to decrypt layer two.

```
let layer2DecryptedStringsVariableName = null;
let layer2DecryptedStrings = null;



const setupLayer2Context = {
  // First, we need to run all function declarations in our context so that we
  // can run the decryption function
  FunctionDeclaration(path) {
    const functionCode = generate(path.node).code
    vm.runInContext(functionCode, layer2Ctx)
  }
}



const findEncryptedStringVisitor = {
   VariableDeclaration(path) {
    const init = path.node.declarations[path.node.declarations.length - 1].init; //
Get initialization of last defined variable



    if (init && init.type === "CallExpression") {
      const args = init.arguments
      if(args.length == 2 && args[0].type === "StringLiteral" && args[1].type ===
"NumericLiteral") {



        layer2DecryptedStringsVariableName = path.node.declarations[0].id.name



        const decryptCode = generate(init).code;
        layer2DecryptedStrings = vm.runInContext(decryptCode, layer2Ctx);
        path.remove() // we are going to replace string references inline, remove
this path
        path.stop() // we've found our decrypted layer 2 variable, stop the visitor
      }
    }
  }
}

traverse(ast_layer2, setupLayer2Context)traverse(ast_layer2,
findEncryptedStringVisitor)
```

Next, we'll need to write a visitor that replaces all references to the variable containing the array of strings with the string itself in-line. Let's take a look at the AST for one of these references:

```
j = O[2];
```

```
  {    "type": "ExpressionStatement",    "expression": {      "type":
"AssignmentExpression",      "operator": "=",      "left": {        "type":
"Identifier",        "name": "j"      },      "right": {        "type":
"MemberExpression",        "object": {          "type": "Identifier",
"name": "O"        },        "computed": true,        "property": {          "type":
"NumericLiteral",          "extra": {            "rawValue": 2,            "raw": "2"
},          "value": 2        }      }    }  }
```

So we're looking for **MemberExpression** nodes and the value of the **property** field, its **NumericLiteral**, used to index the array.

```
const encryptedStringReferencesVisitor = {
  MemberExpression(path) {
    if(path.node.object.name === layer2DecryptedStringsVariableName) {
      stringId = path.node.property.value;
      path.replaceWith(t.valueToNode(layer2DecryptedStrings[stringId]));
    }
  }
}

traverse(ast_layer2, encryptedStringReferencesVisitor)
```

This visitor looks for **MemberExpression** nodes whose **name** field is equal to the name of the strings array variable we identified in the previous visitor. We then use the function **replaceWith()** to replace it with the appropriate string, converted to a node with **valueToNode()**.

Replacing Operators and Identifiers In-Line

Now we're going to write a more complex visitor with some sub-visitors to replace functions that contain simple operators/identifiers with their operator/identifier in-line. Let's take a look at the three different types of functions we'll be looking for:

```
// Unary operators
function cc(a) {
  return -a
}



// Binary operators
function bv(a, c) {
  return a == c
}

// Identifiersfunction bY() {  return window}
```

Unary Operators

First, let's take a look at the unary operator function's AST:

```
  {    "type": "FunctionDeclaration",    "id": {      "type": "Identifier",
"name": "cc"    },    "generator": false,    "async": false,    "params": [      {
"type": "Identifier",        "name": "a"      }    ],    "body": {      "type":
"BlockStatement",      "body": [        {          "type": "ReturnStatement",
"argument": {            "type": "UnaryExpression",            "operator": "-",
"prefix": true,            "argument": {              "type": "Identifier",
"name": "a"            }          }        }      ],      "directives": []    }  }
```

All of our different types of obfuscated operators and identifiers will follow a similar pattern. We're looking for a **FunctionDeclaration** whose **body** type is **BlockStatement**. The **body** of the first item of the **body** of the **BlockStatement** should be of type **ReturnStatement f**or all function types above. Let's go ahead and start creating our visitor:

```
const operatorsIdentifiersVisitor = {  FunctionDeclaration(path) {    const
functionName = path.node.id.name;    if(path.node.body.type === "BlockStatement"
&& path.node.body.body[0].type === "ReturnStatement") {      const returnBody =
path.node.body.body[0]
```

Here we're looking for any and all **FunctionDeclaration** nodes that match the pattern described above. We'll save the first element of the **body** of the **ReturnStatement** in the **returnBody** variable in order to use it later. Next, we want to check the **type** of the **argument** of **returnBody** to see if it's of type **UnaryExpression**. If it is, we'll go ahead and create a new sub-visitor to look for all **CallExpression** nodes that are calling the function containing the unary operator we're trying to deobfuscate. Once the visitor lands on a matching **CallExpression**, we will replace that node with a node that uses the operator directly. The function containing the operator is removed from the AST.

```
      if(returnBody.argument.type === "UnaryExpression") {
        const replaceExpandedUnaryOperatorVisitor = {
          CallExpression(path2) {
            if(path2.node.callee.name === functionName) {
              const inlineOperatorCode =
                    generate({
                      "type": "UnaryExpression",
                      "operator": returnBody.argument.operator,
                      "argument": path2.node.arguments[0] // Use the variable name
the CallExpression uses as an argument
                    }).code
              path2.replaceWithSourceString(inlineOperatorCode);

            }          }        }        traverse(ast_layer2,
replaceExpandedUnaryOperatorVisitor)        path.remove()      }
```

After running the visitor against our code, we will see all calls of functions such as **cc()** above replaced with their corresponding operator in-line:

```
// before
var c = cc(271733879);

// aftervar c = -271733879;
```

Binary Operators

Now we're going to take a look at functions that wrap **BinaryExpression** type nodes. Let's take a look at the **body** of the function **bv()**:

```
{  "body": {    "type": "BlockStatement",    "body": [      {        "type":
"ReturnStatement",        "argument": {          "type": "BinaryExpression",
"left": {            "type": "Identifier",            "name": "a"          },
"operator": "==",          "right": {            "type": "Identifier",
"name": "c"          }        }      }    ],  }}
```

As you can see, a **BinaryExpression** node has a **left**, an **operator**, and a **right**. We can use this to write a visitor that goes to all **CallExpression**
nodes and puts the **operator** in-line:

```
      else if(returnBody.argument.type === "BinaryExpression"
         && returnBody.argument.left.type === "Identifier"
         && returnBody.argument.right.type === "Identifier") {
        const replaceExpandedBinaryOperatorVisitor = {
          CallExpression(path2) {
            if(path2.node.callee.name == functionName) {
              const inlineOperatorCode = generate(
                {
                  "type": "BinaryExpression",
                  "left": path2.node.arguments[0],
                  "operator": returnBody.argument.operator,
                  "right": path2.node.arguments[1]
                }
              ).code;
              path2.replaceWithSourceString(inlineOperatorCode);

            }          }        }        traverse(ast_layer2,
replaceExpandedBinaryOperatorVisitor)        path.remove();      }
```

Functions Returning Identifiers

Finally, we're going to take a look at functions that return identifiers. The function **bY()** above returns only the identifier **window**. Let's take a quick look at the **body** of its AST:

```
{  "body": {    "type": "BlockStatement",    "body": [      {        "type":
"ReturnStatement",        "argument": {          "type": "Identifier",
"name": "window"        }      }    ],    "directives": []  }}
```

As you can see, we simply need to look for functions whose **returnBody** argument type is **Identifier** and then create a sub-visitor to look for all **CallExpression** nodes calling that function and replace them with the **Identifier** node.

```
      else if(returnBody.argument.type === "Identifier") {        const
replaceExpandedIdentifierVisitor = {          CallExpression(path2) {
if(path2.node.callee.name === functionName) {              const
expandedIdentifierCode = generate({                "type": "Identifier",
"name": returnBody.argument.name              }).code;
path2.replaceWithSourceString(inlineIdentifierCode);            }          }        }
traverse(ast_layer2, replaceExpandedIdentifierVisitor)        path.remove()      }
```

After this, we need to close out the visitor **operatorsIdentifiersVisitor** and run it against our layer two AST. Finally, we generate the code for the AST and output it to the console:

```
    }
  }
}
traverse(ast_layer2, operatorsIdentifiersVisitor)

console.log(generate(ast_layer2).code)
```

Final Deobfuscator & Deobfuscated Script

The full script to deobfuscate ScamClub payloads up until this point is as follows. Note that there is more that we could do to beautify this and make it even closer to perfect, but at this point we have a reliable deobfuscator that produces a fully-readable output.

ScamClub Deobfuscation Script
And here's the deobfuscated stage 1 script:

Closing Notes

In this article, we've gone from a fully-obfuscated ScamClub payload to a mostly deobfuscated, easy-to-read script that we can now analyze. There are some other things that can be done to make it even more readable, but for the purposes of understanding what the script does, we've reached a good point.

In the next article in this series, we'll go over the long version of the ScamClub payload, which contains the same obfuscation tricks as well as some new ones. We'll go over the additional things that could make these payloads even more readable, as well as the new tricks the longer version of the payload implements. In another, we will go over the analysis of the deobfuscated payloads including the fingerprinting techniques and exploits used.

I hope this post is helpful in your deobfuscation adventures. For any questions, you can reach me on any of the following:

Email: gregory@confiant.com
Mastodon: @bozoslivehere@ioc.exchange
Matrix: @bozoslivehere:matrix.org