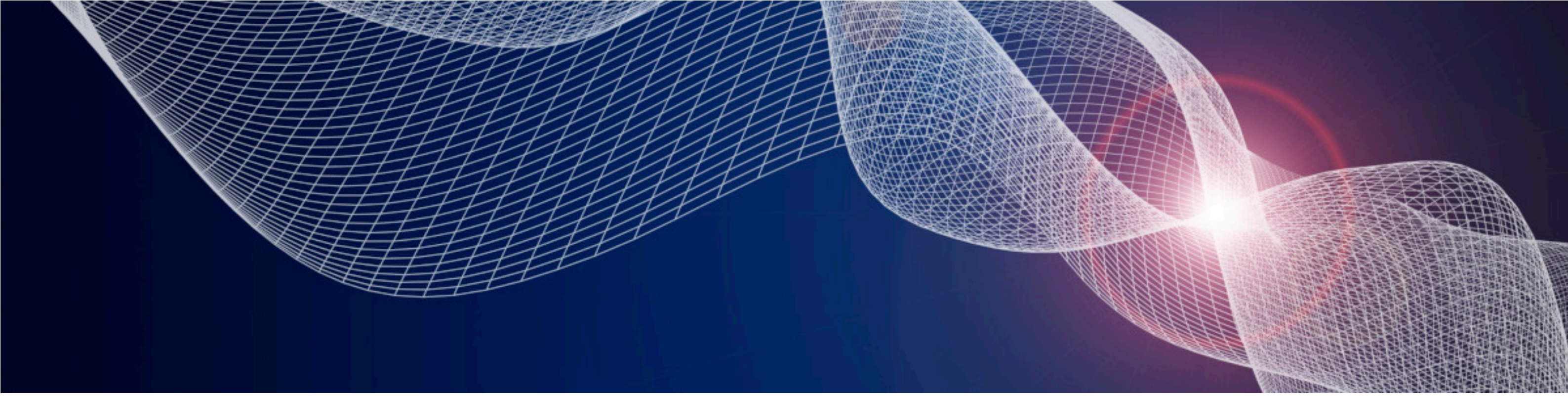

2. Deobfuscation

2.1 Introduction

These days, most programs you get your hands on will have been transformed from the original source code in some way. In most types of programmes, these transformations are not specifically aimed at making the inner workings harder to understand, but serve purposes like minimizing the amount of data transferred via networks (allowing the programs to load faster), optimizing performance or simply making the program executable by a machine.

Some programs, though, are transformed with the express goal of making them harder to understand (a prime example of this of course is malware). These transformations are called *code obfuscation* [1] .

*De*obfuscation then is the art of turning a such transformed program back into source code that can be read, understood, and possibly patched to do things the original programmer didn't intend.

We will first discuss different obfuscation methods, then how to counter them and finally look at tools which can help us deobfuscating code in three different languages: VBA, PowerShell, and JavaScript.

Why these? Well, these languages are often involved in today's attack vectors, e.g. when targeting a windows PC (VBA as entry door via office documents, PowerShell to interact with the outside world) or when attacking unsuspicious visitors of infected or malicious websites.

2.2 Timetable

Topic	Concepts [minutes]	Practice [minutes]	Total [minutes]
Obfuscation	60		60
Analysis	60	360	420
Total	**120**	**360**	**480**

[1] https://en.wikipedia.org/wiki/Obfuscation_%28software%29

2.3 Obfuscation

Good programs are written in a way that makes them easily readable and thus maintainable. How is that achieved? There are a some high level rules to adhere to:

- Choose good names: names which make explanations unnecessary. As an example, a method is probably better named “findNonAdminUsers” than “load”.
- Divide code up in small, self-contained methods. A method loading values from a database, calculating metrics, creating a visualization and exporting that along with raw data to PDF should probably be divided up into different methods doing each step and one top method orchestrating these.
- Group together what belongs together to make it easier to find the next piece of code to read in a control flow.
- Code should be as short as possible (but the above rules take precedence) and not contain redundancy.

In order to understand how to deobfuscate code, we should first understand some techniques commonly used to obfuscate code. What does obfuscation do? It turns well-written - and thus easily readable and understood - programs into very unreadable ones. This is typically done by inverting the rules listed above ;-)

2.3.1 Name Mangling

One of the things that help us most understand code is good naming of variables and methods. So the easiest way of confusing human readers is to arbitrarily change these names to something completely random or even intentionally confusing.

> Name mangling is not always malicious: JavaScript minification shortens all possible names as much as possible as a form of compression which allows to save bandwidth when transferring code between server and clients viewing web content in their browser.

Let’s have a look at an example.

Original

```
method calculateFactorial = (input) => {
  variable result = 1
  variable counter = 1

  loop
    result = result * counter
    exit if counter == input
    counter = counter + 1
  end loop
}
```

Mangled

```
method ydfjkwe823 = (ydfjkwe822) => {
  variable ydfjkwe821 = 1
  variable ydfjkwe820 = 1

  loop
    ydfjkwe821 = ydfjkwe821 * ydfjkwe820
    exit if ydfjkwe820 == ydfjkwe822
    ydfjkwe820 = ydfjkwe820 + 1
  end loop
}
```

The two code snippets do exactly the same - would you agree the original one can more easily be understood?

2.3.2 Data Transformation & Distribution

Even if all variables are renamed, the values assigned to them may still reveal much of what they are meant for. This is even more true for strings which may be output to the user such as prompts to input a value. In order to hide these from human readers, one good approach is to put transformed values into the code and to apply the reverse transformation at runtime. Even legitimate code may have good reasons to hide such strings (e.g. for simple license keys).

Here are some different ways of transforming strings:

- decomposition (e.g. `"We" + "lc" + "om" + "e!"`)
  - String formatting can be used to recompose strings in different orders (e.g. `"{2}{0}{1}"-f "i","!","H"`)
  - Any kinds of operations can be used to compose data (e.g. `5**3//20%5` instead of just `1`)
- transforming them letter by letter
  - simply presenting them in e.g. hex (e.g. `"\\x4f\\x68\\x20\\x6d\\x79\\x21"`)
  - using maths to represent ASCII codes (e.g. `chr(87)+chr(101)+chr(105)+chr(114)+chr(100)+chr(33)`)
  - leveraging language features (e.g. `(![]+[])[!+[]+!+[]]+([![]]+[][[]])[+!+[]+[+[]]] +(![]+[])[+[]]+(!![]+[])[(+!+[]<<+!+[])+!+[]]`)
- adding unnecessary escape characters (e.g. `"di^r c^:^\"`)
- embedding fixed patterns which will first have to be removed (e.g. `"Ha8]ea8]la8]la8]oa8]!"`)
- encoding them (e.g. in base64: `"U2VlPwo="`)
- encrypting them (e.g. `"oi93ssdf2eo23"` - don't waste your time on this ;-))

To add to this, a further complication is to assign different parts of the data to different variables all over the place and to reconstruct them as needed.

2.3.3 Code Bloating

One quite effective deterrent to deobfuscation can be inserting code which doesn't change anything, but greatly increases the amount of code the viewer has to look at. Some data transformation techniques have this effect as well (e.g. letter by letter transformations), so they're often used in combination with pure code bloating techniques.

There are two basic methods used to bloat code:

- Inserting dead code
- Adding identity operations - operations which don't do anything

Both techniques have in common that portions of the code can be removed without any effect on the result.

2.3.3.1 Dead Code Insertion

One very simple and effective way to make code more complex to read is to add control structures where none are required. As a very simple example, we can add code within a conditional block which will never execute:

JavaScript

```javascript
if (!(""+[]) && "false" != !true) {
  console.log("this will run - make sure you understand why!");
} else {
  console.log("this will never run");
}
```

The same concept can of course be used with loops (which run e.g. zero times, are exited early, . . .) and case matching (where only one or even no case matches). There is one additional concept we have to be aware of: `GoTo $label` to jump to a position in code which is labelled `$label`. `GoTo` is not typically taught to programmers these days because it's considered bad style (it's very hard to keep track of what's happening), but it's very effective for exactly that reason!

VBA

```vba
sub foo()
  GoTo aweridfjlk
  ' bloat code
  ' more bloat code
aweridfjlk:
  Debug.Print "this actually happens"
  GoTo weorilksjdf
  ' this doesn't
  ' nothing here to see...
weorilksjdf:
end sub
```

2.3.3.2 Identity Operations

You may remember from maths that there are *identities*. As an example, the *additive identity* for natural numbers is `0`, and their *multiplicative identity* is `1`. The same concept can be applied to different data types, allowing us to pretend to be doing complicated things while not doing anything at all. One common pattern we see is declaring uninitialized variables (or using implicitly declared variables) and concatenating them to strings:

VBA

```vb
asdfle = weofvd + "ll" + dlfier
' ...
werkdr = asasfe + "He" + werlkf
' ...
sdflgkj = owersd + werkdr + flojse + asdfle + asdfke + "o"
```

2.3.4 Control Flow Rerouting

If we want code to be hard to read, we should make sure we make readers jump around as much as possible to keep them from getting a clear picture of what's happening in their minds. Two efficient ways of doing that are

- inlining code: taking code from methods and replacing method calls with the entire code (with the parameter names replaced by the values calculated in the caller method)
- "flattening" control flow by creating a "dispatcher" (usually in the form of case matching), which executes different bits of code based on a variable which represents the next "label" to go to.

Original

```powershell
$a=$args[0]




if($a -eq 0) {







  write-host 1
} else {


  write-host 10



}
```

Obfuscated

```powershell
$a=$args[0]
$b = 0;
while(1) {
  switch($b) {
    0 {
      if($a -eq 0) {
        $b = 1
      } else {
        $b = 2
      }
      break;
    }
    1 {
      write-host 1
      return;
    }
    2 {
      write-host 10
      return;
    }
  }
}
```

2.3.5 Dynamically Executed Code

Many languages can execute code contained in strings. This allows creating code within running code and then executing it, making it harder to understand what's going on in the end. Here are two examples of how code can be run:

PowerShell

```powershell
$c=$ExecutionContext.InvokeCommand.NewScriptBlock("write-host Hi!")
Invoke-Command $c
```

JavaScript

```javascript
eval("console.log(2 + 2)")
```

2.4 Analysis

If possible, use a tool to deobfuscate code you encounter. There are many of them around, and they can usually help you at least with certain parts of the deobfuscation. Since tools are usually dumb, though, we should be able to do this manually as well so we can deal with situations which were not foreseen by the tool authors.

2.4.1 General

Modern IDEs such as Visual Studio Code[2] (VSCode) give us a lot of help when deobfuscating code, for example by showing us unused variables or allowing us to do rename refactorings. If you haven't already, please download and install the application in your working environment.

One thing I would definitely recommend is saving intermediate steps we come up with. The reason is simple: if we make a mistake and e.g. remove code that later turns out not to be "dead", but is dynamically called, we may have to redo many of the steps we had already completed for that "resurrected" piece of code.

A very simple way of doing that is to initialize a git repository in the folder you do your work. If you're not familiar with Git, here's[3] a very short intro. Once you've got a git shell installed on your system, you can follow these steps (in your working directory):

1. `git init`
2. `git add .`
3. `git commit -m "Initial"`
4. Peel away an obfuscation layer
5. `git commit -m "some message"`
6. Not yet done? Go to (4)

You can also implicitly save intermediate steps by using e.g. the Local History[4] extension for *VS Code*.

An alternative to that approach is to script the deobfuscation steps, e.g. in Python - or using specific tools. One very well-known general-purpose tool hey call *The Cyber Swiss Army Knife* is CyberChef[5]. It lets you create "recipes", which allows you to daisy-chain operations transforming inputs. You can either use the tool directly online, download it to your machine or use one of the available docker images (e.g. this one here[6]).

[2] https://code.visualstudio.com/
[3] https://guides.github.com/introduction/git-handbook/
[4] https://marketplace.visualstudio.com/items?itemName=xyz.local-history
[5] https://gchq.github.io/CyberChef/
[6] https://github.com/mpepping/docker-cyberchef/

2.4.1.1 Steps

There's not a generic approach to deobfuscating that will be the most straight forward in every case. We can generally say that if we aim to quickly reduce the amount of code we're dealing with, we will likely save a lot of time. I would recommend the following basic "cookbook":

1. Run a code formatter
2. Unbloat
    1. Remove dead code
    2. Remove identity operations
3. Compact & clean up data
    1. Evaluate constant mathematical expressions
    2. Apply constant string concatenations
    3. Inline data from other sources
    4. Decode / decrypt

You may have to do certain steps more than once if a later transformation re-introduces elements that were cleaned up previously (e.g. if dynamic code is introduced which again has been obfuscated).

At any point, it can help to rename method names or variables which we recognize for what they are. Use your IDE's refactoring capabilities to do this!

2.4.2 VBA

The first step to understanding VBA code inside office documents is to get access to the code - without exposing ourselves to the risk of executing malicious code. As always, there are lots of ways of doing so - we will use a tool written in Python to do it: olevba [7] , which is part of the oletools [8]) suite.

The official installation instructions ask you to execute `sudo -H pip install -U oletools`, but that seems a bit risky, so we'll take the safer path:

1. Make sure you have pipenv installed [9] (`which pipenv`)
2. Create a separate folder somewhere (e.g. `~/reversing/oletools`) and `cd` into it
3. Run `pipenv install oletools` - this will install oletools into a user specific virtual environment

With this out of the way, from the folder we created, we can now run the different tools in the *oletools* suite, e.g. `pipenv run olevba $\$$FILE` to extract VBA code from an office document.

Time to exercise!

2.4.2.1 Automated Deobfuscation

There currently doesn't seem to be much of an alternative to ViperMonkey [10] . The program implements its own VBA parser in Python and is pretty successful at deobfuscating code. The downside is that of course the more dedicated malware authors know that the parser is not complete and thus find ways to crash it.

In order to simplify using the program, I've created a fork here [11] in which I've added a Dockerfile which works with an adjusted `dockermonkey.sh` (and of course have submitted a pull request). To use it, simply clone the repository and run `./docker/dockermonkey.sh $maldoc`, with `$maldoc` the document you want to deobfuscate.

The nice thing about this solution is that `dockermonkey.sh` automatically disconnects the docker container's network, thus effectively creating an isolated sandbox for analysis.

> `ViperMonkey` uses `olevba` in the background, so no need to extract the code manually.

Time to exercise!

[7] https://github.com/decalage2/oletools/wiki/olevba
[8] https://github.com/decalage2/oletools
[9] https://pipenv.kennethreitz.org/en/latest/install/#installing-pipenv
[10] https://github.com/kirk-sayre-work/ViperMonkey
[11] https://github.com/chezwicker/ViperMonkey

2.4.3 PowerShell

When looking at obfuscated PowerShell code, you're most likely not dealing with something created by a benevolent programmer, so I would recommend you do any work in a virtual machine. You can get a Windows VM from here [12] .

2.4.3.1 Language

If you don't know *PowerShell*, but have mastered *Bash* syntax, you're in luck: the two are rather similar, so if you can read bash syntax, there's no reason you should struggle with *PowerShell*.

Let's look at some differences between the languages so you can hit the ground running.

Area	Bash	PowerShell	Comment
String comparison	`=`, `!=`	`-eq`, `-ne`	
Logical operators	`-o`, `-a`	`-or`, `-and`	
Negation	`! $expr`	`! ($expr)`	
Empty	`-z $var`	`$var -eq $null`	
File existence	`-e $file`	`Test-Path $file`	
Escaping	`\`	`` ` ``	Backtick
Printing	`echo $text`	`Write-Host $text`	
Variable assignment	`foo="bar"`	`Set-Variable foo "bar"`	Also `set foo "bar"` or simply `$foo="bar"`
Path	`$PATH`	`$env:path`	

Control Structures read almost the same, but the keywords surrounding the code blocks are replaced by braces, e.g.

Bash

```bash
if [ condition ]; then
  # ...
elif [ condition ]; then
  # ...
else
  # ...
fi
```

PowerShell

```powershell
if (condition) {
  # ...
} elseif (condition) {
  # ...
} else {
  # ...
}
```

One useful feature you'll likely find "abused" in obfuscated code is string formatting using "-f", described e.g. here [13] . Here's an example:

PowerShell

```powershell
"{1}, {0}" -f "there!","Hi" # => Hi, there!
```

Time to exercise!

[12] https://developer.microsoft.com/en-us/windows/downloads/virtual-machines/
[13] https://ss64.com/ps/syntax-f-operator.html

2.4.3.2 Automated Deobfuscation

A tool which is rather nice is PSDecode [14] - I've also created a fork to help us a bit further along with the deobfuscation of the sample I chose. You can follow the simple installation instructions in the `README.md` file to set it up.

This time, I highly recommend you use a VM (or a windows docker instance if you are working on Windows) and to disconnect all access to your host machine when running the tool.

Time to exercise!

[14] `https://github.com/chezwicker/PSDecode`

2.4.4 JavaScript

JavaScript often serves to deliver malware as well, often via attachments to email opened by unsuspecting recipients. Like VBA, the code executed very often downloads the actual payload and executes it via *PowerShell*, targeting *Windows* environments.

A common entry point to the program delivered is an *immediately invoked function expression*. This works by defining a function immediately followed by the arguments it will be passed, e.g.

JavaScript

```
(function x(a) {
  console.log(a);
})(1)
```

Three features of JavaScript which are very important in (de)obfuscation are property access by name and the comma operator. Let's make sure we understand these!

- In an object `x = { foo: 1, bar: true }` we can access property `bar` by using `x["bar"]`. What may not be completely obvious at first is that this also works for methods. As an example, we an access the method `push` of an array `y` be using `y["push"](2 )` to append the value `2` to `y`. And since we've seen that there are lots of ways to obfuscate strings, this is a great opportunity for control flow obfuscation.
- Anything inside parentheses is an expression. We've seen it used just now, when we wrapped the definition of function `x(a)` above in parentheses to return an executable function!
- The comma operator can be used in an expression to evaluate each operand from left to right and to return the result of the last one.

JavaScript

```
x = (console.log("Hi!"), "I'm Bob");
console.log(x);
```

And maybe one more thing of importance that wouldn't be used in clean code: one can declare (and assign) multiple variables following one `var`, `let`, or `const` keyword. The following two code blocks are equivalent

One-Line

```
var x, y = 2, z = (1,2,3);
```

Multi-Line

```
var x
var y = 2
var z = (1,2,3);
```

Time to exercise!

2.4.4.1 Tooling

There's particularly one tool which is useful and actively maintained: box-js [15] . It rewrites the source code to make it easier to read and, importantly, executes it in an emulated Windows JScript environment.

You can also run the analysis in a *Docker* container using the following command, with

- `$hostLocalDir` the path to the directory on your host (VM) containing the JavaScript file(s) to analyze - this is mounted into the docker container at location `/samples`.
- `$dirOrFile` the directory or file to analyze. This will be relative to `/samples`, which is the folder within the container to which the `$hostLocalDir` is mapped.

Bash

```bash
docker run \
  --rm \
  --volume $hostLocalDir:/samples \
  capacitorset/box-js \
  box-js $dirOrFile \
  --output-dir=/samples \
  --loglevel=debug
```

–volume $hostLocalDir:/samples
capacitorset/box-js
box-js $dirOrFile
–output-dir=/samples
–loglevel=debug \end{code}

Time to exercise!

[15] https://github.com/CapacitorSet/box-js