

# Obfuscation

---

## What and Why?

Obscures internals of a program

Perfect obfuscation not practical with todays tech, only possible with cloud computing when nobody has access to the program

Every obfuscated program will be reversed eventually

## Obfuscation <-> Clean Code

Clean Code is the opposite of source obfuscation

During .com boom MANY clean code guidelines were made (Sun Microsystems)

Some seniors not satisfied with quality of juniors work

By making something unmaintainable, you could make sure you wont get laid off (theoretically)

## Example 1: Bad Java Code

- non-descriptive names
- unicode characters in names
- Anti-Patterns (Reflection)

Doesnt use control-flow obfuscation because I want to keep snippet short

Fun fact: Java ignores non-printable characters

## What is enough?

---

Race between obfuscation and deobfuscators

Really no need to obfuscate more if antivirus doesnt detect

Overdoing it just gives away techniques that might have been beneficial in the future

Consumers dont like overprotective measures

## Overdoing it, A LOT

CTFs use very weird, not-relevant tactics

Contest is only about obfuscating the source, rules are updated to keep it interesting (<https://www.ioccc.org/>)

Code golf results in very weird & hard to read code

## C Donut

We all know it

Would be very hard to read if we didnt know what it is

- Magic numbers
- One character names
- Useless characters

## Language dependent

Different obfuscation techniques based on if you have the source

Vastly different techniques depending on platform and language

## Interpreted languages

---

Interpreted Languages usually don't contain actual exploit, rather just a downloader, so obfuscation is usually not too strong

JS Frameworks output horrible code

Powershell or VBS is usually only lightly modified because it is not the actual payload

## Encoding

Taking part of or entire code and encoding it, usually with Base64

More signatures to track for antivirus

## Variable and Identifier Transformation

Change names of variables, usually into random symbols

Hide values of constants

Use functions to hide values and maybe even names of other functions

## Example 2: JS Obfuscator

Not a lot of effort

All Strings are wrapped in a function

Can sometimes still be understood, but can just be obfuscated more than one time

## Compiled Languages

---

Java and C# let you recover the source (relatively well), while still using a compiler

The previous techniques are completely valid for these two and at least somewhat usable for all other languages

Actual applications / malware are usually compiled, so better obfuscation is possible

## String Encryption

---

Plain Strings are easy to read, remember "Reversing with Lena"

Strings are encoded/encrypted just like in interpreted languages

Enterprise Dotfuscator can do it

(Dotfuscator will show up a lot, because companies used to love C# Apps)

## Control Flow Obfuscation

---

Adding more branches

Makes Graph view far more complicated

### Obfuscated Function Graph

- Never used code
- Code that does nothing relevant
- One time loops
- more cases than necessary

## High-End Tools

Basic Control Flow Obfuscation can be "solved" automatically

Tool searches for them and eliminates simple Control Flow Obfuscation, or just run it and trace the execution

## Control Flow Flattening

Very widely used

Also fairly high effort to reverse

But not that hard to do automatically

*show how it works*

State could be set as a side-effect somewhere

## Opaque Predicates

"Impossible" for static analysis because it needs external knowledge, return values for API calls for example

Dynamic analysis can help, but also fairly high effort

## Tools

These tools can do the past few things

Dexguard basically does everything for android

Tigress is free obfuscator for binaries

Dotfuscator free version is very bad

## How to Rev?

Execution Trace/Dynamic Analysis: run it and see what happens

Find the obfuscated points and patch them

Try to manually reconstruct the original program

## Packing

---

Taking the (bytecode) of a function

Encrypt / encode it

Putting it somewhere else

That thing is then decrypted during runtime

Google play seems to have issues with detecting packed malware

## VM Obfuscation

Subset of packing

Different interpreter bundled to executable, with instructions somewhere

Might not be traditional bytecode, basically anything is possible

Existing interpreters can be bundled to a file

pyinstaller bundles a python interpreter with python bytecode

## Tools

UPX very well known and free

Tigress can do it

pyinstaller packs python bytecode (.pyc)

## How to Rev?

Entropy analysis could be used

1. Find where it is unpacked (some sort of loop that references data)

2. After the loop is a jump somewhere (OEP)
3. Break there during execution or manually unpack

Fairly High Effort

## Novelty Techniques

---

Too impractical to use in real-world

Not completely useless because it can discover interesting behaviour

### Instruction Overlapping

Instructions may contain other valid instructions in their bytecode

If an instruction has a number as an argument, that number can be chosen so that it is another instruction

By doing that with relative jumps you can hide parts of the program

With enough work, a separate execution path can be created

Insane amount of work to do on a useful scale

### Flow Graph Art

Using control flow obfuscation on such a scale that IDA shows an image in the graph

Images NEED to be small and greyscale

Works better with high contrast

REpsych doesn't actually obfuscate, the code is not in the picture

Picture

It is the KaindorfCTF logo

Works well enough because of brightness difference

### Language Features

Syntactic Sugar easily turns into unreadable code

If few people know how to use a feature, even less will be able to understand abuse

Anything can become unreadable if overdone enough

## Demo

---

Files here: <https://github.com/Reapie/NVSS-Obfuscation>

# Powershell Obfuscation

---

There is an insane amount of tools out there, most of them do basically the same thing.

PowerShell even has some builtin obfuscation option with `-enc` which will decode a base64 String. But when using this method it is important to encode the text to Unicode Base64, otherwise the decode will fail:

```
$Text = 'Start-Process "https://www.youtube.com/watch?v=dQw4w9WgXcQ" '  
$Bytes = [System.Text.Encoding]::Unicode.GetBytes($Text)  
$EncodedText =[Convert]::ToBase64String($Bytes)  
$EncodedText
```

The "Invoke-Stealth" (<https://github.com/JoelGMSec/Invoke-Stealth>) tool is just one of many, and the created file "obfuscated.ps1" was created with the following command:

```
pwsh Invoke-Stealth/Invoke-Stealth.ps1 obfuscated.ps1 -technique ReverseB64
```

It does the obfuscation in-place, thus "obfuscated.ps1" as the first argument.

These are just examples for simple encodings but they are still effective as they prevent humans from recognising the purpose of the script quickly.

Even with this harmless sample, my antivirus prevents execution, showing that Base64 is very common in real-world attacks.

## Control Flow

---

This is a simple, manually made example of control-flow obfuscation. Commercial obfuscators are way better but they usually don't have free trial versions as that would let basically anybody create well obfuscated malware, this is why I just did it manually.

It uses multiple techniques:

- Control Flow Flattening - Global State Variable used in switch hides the execution order
- (Slight) Opaque Predicates - Argc will always be  $\geq 1$  but decompilers don't have this knowledge programmed into them
- Dead Code - There are comparisons and data that are never used

To make sure that the compiler doesn't optimise anything away, make sure to compile with no optimisation (`-O0`)

The "original" main function is commented out, but the difference in readability is obvious.

Because the obfuscation is very basic, it can easily be solved using symbolic execution (`solve.py`)

# UPX Packer

---

One again, enterprise products are way out of my budget. UPX has a built in unpacker and signs binaries it packs (`strings simple_packed | grep UPX`).

The input for the packer was once again `../simple_program`.

Looking at the binary in some Reverse Engineering Tool reveals how good Packers are, but UPX has a builtin decompress function. To use it just run `upx -d <file>`, but make sure you don't run it on the original file because it decompresses in-place.

# MoVfuscator

---

<https://github.com/xoreaxeaxeax/movfuscator>

<https://www.youtube.com/watch?v=R7EEoWg6Ekk>

xoreaxeax does awesome stuff and has many interesting talks, his MoVfuscator project makes use of a "basic" building block to turn C code into only MOV instructions.

[https://www.xorpd.net/pages/xchg\\_rax/snip\\_2c.html](https://www.xorpd.net/pages/xchg_rax/snip_2c.html)

```
mov    qword [rbx + 8*rcx], 0
mov    qword [rbx + 8*rdx], 1
mov    rax, qword [rbx + 8*rcx]

mov    qword [rbx], rsi
mov    qword [rbx + 8], rdi
mov    rax, qword [rbx + 8*rax]
```

This piece of code is the basis of the MoVfuscator and moves rsi or rdi into rax depending on whether rcx and rdx are different or not. It is equivalent to this ternary construct: `rax = (rcx == rdx) ? rdi : rsi`

As already mentioned, it is pure novelty and is basically never used in real-world applications but still very interesting.

The input file for this sample was: `../simple_program.c`

# List Comprehension

---

Python has a lot of features to get "pythonic" solutions to problems, on its own they are quite acceptable but you can definately overdo it.

This example does exactly that, overdo list-comprehensions, use lambdas, inline everything and use horrible variable naming.

Even though this example uses python, as already said, every language has some specific features but some AVX C is not as good for demonstration purposes as python.