Code Obfuscation $10^{**2} + (2*a + 3) \% 2$
Gaetan Ferry

@mabo^W  Not on twitter

Security expert @Synacktiv :
  - Offensive security company : pentest, red team, reverse/exploit…

Pentest team peon:
  - 1 me / 17 pentester / 41 ninjas
  - Breaking things since 2012
  - Web, internal, external, IOT, indus, cloud
WE WANT YOU FOR OUR NINJA ARMY!
INTRODUCTION
Why this presentation?

Why not?

Obfuscation is an undervalued domain
- Usefulness often discussed
- Defined as security by obscurity

Therefore abandoned

Therefore unknown
→ We want to redeem obfuscation
What is in this presentation?

- Objectives of a proper obfuscation
- Details of classic obfuscation patterns
- Implementation with / for Python
- Examples and ...

...demos (pray demo gods)
WHAT IS OBFUSCATION?
A bit of theory

Let $P$ be the set of all programs and $T$ a set of transformations such as:

$$T_i : P \rightarrow P$$

$T_i$ is an obfuscation transformation if and only if:

- $\text{out}(T_i(P_k)) = \text{out}(P_k)$
- analysis of $T_i(P_k)$ is harder than analysis of $P_k$

$T_i$ is considered efficient if the knowledge of $T_i(P_k)$ is equivalent to having a black-box oracle of $P_k$. 
A bit of theory

Let $P$ be the set of all programs and $T$ be a set of transformations such as:

$$T_i : P \rightarrow P$$

$T_i$ is an obfuscation transformation if and only if:

- $\text{out}(T_i(P_k)) = \text{out}(P_k)$
- analysis of $T_i(P_k)$ is harder than analysis of $P_k$

$T_i$ is considered efficient if the knowledge of $T_i(P_k)$ is equivalent to having a black-box oracle of $P_k$. 
This is not what we are doing

- Obfuscation does not stand well theory
- Theoretical results are demoralizing
  - In general cases obfuscation is impossible
  - Some exceptions: point functions

- Let's go with a more pragmatic approach
A more pragmatic approach

- **Process of complicating programs**
  - Take a beautiful well written program
  - Transform it in some way
  - Retrieve an obscure ugly program

- **Two rules to follow**
  - Resulting program is semantically equivalent
  - More difficult to analyze and understand
A more pragmatic approach

- Process of complicating programs:
  - Take a beautiful well written program
  - Transform it in some way
  - Retrieve an obscure ugly program

- Two rules to follow:
  - Resulting program is semantically equivalent
  - More difficult to analyse and understand
Why do you want obfuscation?

- Useful for good and bad guys
Why do you want obfuscation?

- Useful for good and bad guys

Protect industrial secrets
Discourage hackers who open the thing
Why do you want obfuscation?

- Useful for good and bad guys
- Protect industrial secret
- Discourage hackers who open the thing
- Bypass sandbox / antivirus detection
- Prevent reverse engineering by the good guys
LET'S OBFUSCATE THINGS
How to complicate a program?

- Remove as much information as possible

- Three main directions:
  - Abstractions
  - Data
  - Control flow

- We need to obfuscate each kind
How to complicate a program?

```python
import struct
import binascii
import math

lrot = lambda x, n: (x << n) | (x >> (32 - n))

class MD5:
    A, B, C, D = (0x07452301, 0xefcdab89, 0x98badcfe, 0x098badcfe)

    # r specifies the per-round shift amounts
    r = [7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 4, 11, 16, 23, 4, 11, 16, 23, 6, 10, 15, 21, 6, 10, 15, 21]

    # Use binary integer part of the sines of integers (Radians) as constants
    k = [int(math.floor(abs(math.sin(i + 1)) * (2 ** 32))) for i in range(64)]

def __init__(self, message):
    length = struct.pack('B', len(message) * 8)
    while len(message) > 64:
        self.handle(message[:64])
        message = message[64:]
    message += '\x80'
    message += '\x00' * ((56 - len(message) % 64) % 64)
    message += length
    while len(message):
        self.handle(message[:64])
        message = message[64:]
```

Abstractions
Data
Control flow
LET'S OBFUSCATE ABSTRACTIONS
Program abstractions

- Abstractions help understand programs
  - Imagine a program without proper function names or convoluted class hierarchy!
- Giveaway much of the program semantic
  - Division in semantic blocks
  - Role of the blocks
- Sensitive abstractions:
  - Variables
  - Functions
  - Classes
Names obfuscation

- First step of a successful obfuscation
  - Remove meaningful names from the code
  - Replace with random or unrelated ones

- This information is unrecoverable! \o/

- EZ as 123:
  - Search for all declarations functions, variables, class
  - Replace at each usage location
def power(number, exponent) {
    count = number
    while (exponent > 1) {
        count = count * number
        exponent = exponent - 1
    }
    return count
}

def toast(number, exponent) {
    count = number
    while (exponent > 1) {
        count = count * number
        exponent = exponent - 1
    }
    return count
}

def toast(bread, butter) {
    salad = bread
    while (butter > 1) {
        salad = salad * bread
        butter = butter - 1
    }
    return salad
}
Going further

- Does not seem sufficient
  - Still leaking information
  - Program partitioning unchanged

- We should try to break things

- Ideas:
  - Function inlining
  - Merging / Splitting

- Warning: Beware of introspection calls!
def add(a, b):
    count = a
    while (b > 0):
        count += 1
        b -= 1
    return count

def mult(a, b):
    count = 0
    while (b > 0):
        count += a
        b -= 1
    return count

A = 2
B = 3
C = add(A, B)
D = mult(C, A)
```
def add(a, b):
    count = a
    while (b > 0):
        count += 1
        b -= 1
    return count

def mult(a, b) {
    count = 0
    while (b > 0){
        count += a
        b -= 1
    }
    return count

def toast(a, b, c, d, e) {
    if (e) {
        count = a
        while (b > 0){
            count += 1
            b -= 1
        }
        return count
    } else {
        count = 0
        while (d > 0){
            count += c
            d -= 1
        }
        return count
    }
}

A = 2
B = 3
C = add(A, B)
D = mult(C, A)

A = 2
B = 3
C = toast(A, B, B, A, true)
D = toast(A, C, C, A, false)
```
Function merging - smarter

```python
def add(a, b):
    count = a
    while (b > 0):
        count += 1
        b -= 1
    return count

def mult(a, b):
    count = 0
    while (b > 0):
        count += a
        b -= 1
    return count

def toast(a, b, e):
    if (e):
        count = a
        add = 1
    else:
        count = 0
        add = a
    while (b > 0):
        count += add
        b -= 1
    return count

A = 2
B = 3
C = add(A,B)
D = mult(C,A)
C = toast(A,B,True)
D = toast(C,A,False)
```
(disappointing) DEMO

OBfuscate!
Before

```python
import struct
import binascii
import math

# lrot = lambda x, n: (x << n) | (x >> (32 - n))

class MD5:
    A, B, C, D = (0x67452301, 0xefcda6b9, 0x98badcfe, 0x10325476)
    # r specifies the per-round shift amounts
    r = [7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20,
        4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21]
    # Use binary integer part of the sines of integers (Radians) as constants
    k = [int(math.floor(abs(math.sin(i + 1))) * (2 ** 32)) for i in range(64)]

def _init_(self, message):
    length = struct.pack('<Q', len(message) * 8)
    while len(message) > 64:
        self._handle(message[:64])
        message = message[64:]
    message = message[64:]
    while len(message):
        self._handle(message[:64])
        message = message[64:]

    def _handle(self, chunk):
        w = list(struct.unpack('<' + 'I' * 16, chunk))
        a, b, c, d = self.A, self.B, self.C, self.D
        for i in range(64):
            if i < 16:
                f = (b & c) | ((~b) & d)
                g = i
            elif i < 32:
                f = (d & b) | ((~d) & c)
                g = (5 * i + 1) % 16
            elif i < 48:
                f = b & c | (d & ~c)
                g = (3 * i + 5) % 16
            else:
                f = c & (b | (~d))
                g = (7 * i) % 16
            A = D + (E - A - f) - k[i] + w[i] + (math.sin((i + 1) * math.pi / 2))
            B = A + E - D
            C = B + A - D
            D = C + A - B
```

After

```python
import struct
import binascii
import math

# lrot = lambda x, n: (x << n) | (x >> (32 - n))

class MD5:
    A, B, C, D = (0x67452301, 0xefcda6b9, 0x98badcfe, 0x10325476)
    # r specifies the per-round shift amounts
    r = [7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20,
        4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21]
    # Use binary integer part of the sines of integers (Radians) as constants
    k = [int(math.floor(abs(math.sin(i + 1))) * (2 ** 32)) for i in range(64)]

def _init_(self, message):
    length = struct.pack('<Q', len(message) * 8)
    while len(message) > 64:
        self._handle(message[:64])
        message = message[64:]
    message = message[64:]
    while len(message):
        self._handle(message[:64])
        message = message[64:]

    def _handle(self, chunk):
        w = list(struct.unpack('<' + 'I' * 16, chunk))
        a, b, c, d = self._state'
        for i in range(64):
            if i < 16:
                f = (b & c) | ((~b) & d)
                g = i
            elif i < 32:
                f = (d & b) | ((~d) & c)
                g = (5 * i + 1) % 16
            elif i < 48:
                f = b & c | (d & ~c)
                g = (3 * i + 5) % 16
            else:
                f = c & (b | (~d))
                g = (7 * i) % 16
            A = D + (E - A - f) - k[i] + w[i] + (math.sin((i + 1) * math.pi / 2))
            B = A + E - D
            C = B + A - D
            D = C + A - B
```
LET'S OBFUSCATE DATA
Program data

- All programs contain data:
  - Numbers, strings, arrays, etc
- Often (always) discloses important information:
  - Status / debug messages
  - Important constants (MD5, AES S-Box, etc)
- We want to hide those nasty values!
  - In our example: integers
Program data

- All programs contain data:
  - Numbers, strings, arrays, etc

- Often (always) discloses important information:
  - Status / debug messages
  - Important constants (MD5, AES S-Box, etc)

- We want to hide those nasty values!
  - In our example: integers

USE OPAQUE PREDICATES!
Opaque predicates and values

- One of the core concepts of obfuscation
- We want to build expressions for which:
  - Value is known at obfuscation time
  - At run time value is hard to determine
- When value is a boolean it's a predicate
Opaque predicates – naive idea

I KNOW SOMETHING HARD

MATHEMATICS
Opaque predicates – naive idea

- Open a mathematics course book
- Ctrl + F “demonstrate that“
- Profit

Examples:
- $(n^2 + n) \mod 2 = 0$
- If $n$ is odd : $n^2 \mod 8 = 1$
- $(3 \cdot (2n + 2) + 1) \mod 8 = 2$
Opaque predicates – naive idea

```python
def add(a, b):
    count = a
    while (b > 0):
        count += 1
        b -= 1
    return count
```
Opaque predicates – naive idea

```python
def add(a, b):
    count = a
    while (b > 0):
        count += 1
        b -= 1
    return count
```

\[(n^2 + n) \mod 2 = 0\]
\[n^2 \mod 8 = 1\]
Opaque predicates – naive idea

```python
def add(a, b):
    count = a
    n = rand()
    while (b > (n² + n) % 2):
        count += n² % 8
        b -= n² % 8
    return count
```

\[(n^2 + n) \mod 2 = 0\]
\[n^2 \mod 8 = 1\]
Opaque predicates – naive idea

```python
def add(a, b):
    count = a
    while (b > 0):
        count += 1
        b -= 1
    return count
```

\[(n^2 + n) \mod 2 = 0\]
\[n^2 \mod 8 = 1\]

```python
def add(a, b):
    count = a
    n = rand() * 2 + 1
    while (b > (n^2 + n) \mod 2):
        count += n^2 \mod 8
        b -= n^2 \mod 8
    return count
```

???
Opaque predicates – naive idea

```python
def add(a, b):
    count = a
    while (b > 0):
        count += 1
        b -= 1
    return count
```

\[(n^2 + n) \mod 2 = 0\]
\[n^2 \mod 8 = 1\]

```python
def add(a, b):
    count = a
    n = rand() * 2 + 1
    while (b > (n^2 + n) \mod 2):
        count += n^2 \mod 8
        b -= n^2 \mod 8
    return count
```

```python
def add(a, b):
    count = a
    while (b > (a^2 + a) \mod 2):
        n = count*2+1
        count += (n)^2 \mod 8
        b -= (n)^2 \mod 8
    return count
```
Problem: Smart cat is smart! Smart cat knows mathematics!

Attacking those predicates is easy:
- Build a collection of mathematics results
- Pattern match known relations
- Replace

We can do better
Array aliasing

- Let's build our own mathematical results
  - Create an array
  - Decide properties
  - Initialize the array respecting the properties

- Then use the properties like previously
Array aliasing

Example:

\[
A = [17, 53, 3, 5, 1, 8, 25, 33, 4, 1]
\]

\[
== 3 \text{ mod } 5
\]

\[
== 1 \text{ mod } 4
\]

Array aliasing

```python
def add(a, b):
    count = a
    while (b > 0):
        count += 1
        b -= 1
    return count
```

1 == A[4]%A[8]
1 == A[0]%A[8]
Array aliasing

```python
def add(a, b):
    count = a
    while (b > 0):
        count += 1
        b -= 1
    return count
```

```python
A = [17, 53, 3, 5, 1, 8, 25, 33, 4, 1]
def add(a, b):
    count = a
        b -= A[0]%A[8]
    return count
```
Array aliasing

- Still insufficient:
  - Global array is static
  - Attacker can globally replace values

- We need to bring indecision in!
  - Idea: change the array during the program’s execution
  - Hard! (e.g. How to know the state in function bodies?)
Array aliasing

- Still insufficient:
  - Global array is static
  - Attacker can globally replace values

- We need to bring indecision in
  - Idea: change the array during the program’s execution
  - Hard! (e.g. How to know the state in function bodies?)
  - But not if you keep the properties
Array aliasing

Example:

\[ A = \begin{bmatrix} 17, & 53, & 3, & 5, & 1, & 8, & 25, & 33, & 4, & 1 \end{bmatrix} \]

Array aliasing

- Example:

\[
A = [17, 53, 3, 5, 1, 8, 25, 33, 4, 1]
\]

\[
A = [25, 58, 3, 5, 5, 33, 17, 8, 4, 1]
\]

Array aliasing

```python
def add(a, b):
    count = a
    while (b > 0):
        count += 1
        b -= 1
    return count
add(2, 3)
```

```python
A = [17, 53, 3, 5, 1, 8, 25, 33, 4, 1]
def add(a, b):
    count = a
    while (b > 0):
        b -= A[0] % A[8]
    return count
add(2, 3)
```

1 == A[0] % A[8]
Array aliasing

```python
def add(a, b):
    count = a
    while (b > 0):
        count += 1
        b -= 1
    return count

add(2, 3)
```

```python
A = [17, 53, 3, 5, 1, 8, 25, 33, 4, 1]
def add(a, b):
    count = a
        b -= A[0]%A[8]
    }
    return count

A = [17, 53, 3, 5, 1, 8, 25, 33, 4, 1]
def add(a, b):
    count = a
        b -= A[0]%A[8]
    }
    return count

add(2, 3)
```
Array aliasing

Results

- Data now changes at each run
- Function add change
- Guessing the value of add(2,3) now requires analyzing more than just the add function
- Result might change at each call

```python
A = [17, 53, 3, 5, 1, 8, 25, 33, 4, 1]
def add(a, b):
    count = a
        b -= A[0]%A[8]
    }
    return count
c = add(2,3)
D = add(2,3)
C == D ???
```
Array aliasing

Results

Data now change at each run

Function add change

Guessing the value of add(2,3)

now require analyzing more

than just the add function

Result might change at each call

A = [17, 53, 3, 5, 1, 8, 25, 33, 4, 1]

def add(a, b):
    count = a
        b -= A[0] % A[8]
    return count

c = add(2, 3)
D = add(2, 3)
C == D ???
import struct
import binascii
import math

lrot = lambda x, n: (x << n) | (x >> (32 - n))

class MD5:
    A, B, C, D = (0x67452301, 0xefcda89, 0x98badcfe, 0x10325476)

    # r specifies the per-round shift amounts
    r = [7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21]

    # Use binary operation of the sines of integers (Radians) as constants
    k = [int(math.floor(abs(math.sin(i + 1) * (2 ** 32)))) for i in range(64)]

    def __init__(self, message):
        self._message_length = struct.pack('!Q', len(message) * 8)
        message = message[64:]
        self._message = message[64:]
        self._message_length = len(message)
        self._message_length = message[64:]

    def _handle(self, chain):
        a, b, c, d = self._a, self._b, self._c, self._d

        for i in range(64):
            if i < 16:
                f = (b & c) | ((~b) & d)
            elif i < 32:
                f = (d & b) | ((~d) & c)
            elif i < 48:
                f = b & c & d
            else:
                f = c & (b | (~d))

            g = (3 * i + 3) % 16
            h = (5 * i + 1) % 16
            k[i] = struct.unpack('!I', chain)[0]
            A = (A + f(b, c, d) + k[i] + lrot(A, r[i]) + a) & 0xFFFFFFFF
            B = (B + f(a, b, c) + k[i + 1] + lrot(B, r[i + 1]) + b) & 0xFFFFFFFF
            C = (C + f(b, c, d) + k[i + 2] + lrot(C, r[i + 2]) + c) & 0xFFFFFFFF
            D = (D + f(a, b, c) + k[i + 3] + lrot(D, r[i + 3]) + d) & 0xFFFFFFFF

            a = A
            b = B
            c = C
            d = D

After
LET'S OBFUSCATE CONTROL FLOW
Control Flow Graph

- All programs make use of control instructions
  - if, while, for, switch, etc
- They define a “Control Flow Graph”
  - Composed of test and instructions blocks
  - Define which instruction is executed when
  - Wise attacker can deduce information of the CFG
- We want to obfuscate that
N = RAND()
C = 2
if (N % 2 == 0) {
    while (N > 0) {
        C = C * C
        N = N - 1
    }
} else {
    C = 0
}
print(C)
Control Flow Graph – Naive (?)

- Ideas:
  - Add dead branches
  - Duplicate branches

- Increases the amount of code to analyze

→ Use opaque predicates!
N = RAND()
C = 2

N \% 2 == 0

N > 0

C = C * C
N = N - 1

C = 0

print(C)

N = RAND()
C = 2

N \% 2 == 0

N > 0

C = C² / C
N = N - 1

C = 1

C = 0

print(C)

N = -(- N + 1)
Control Flow Graph - Example

N = RAND()
C = 2

N % 2 == 0

N > 0

C = C * C
N = N - 1

N % 4 == 0

C = 0

C = 1

N = -(N + 1)
C = C³ / C

print(C)

N = RAND()
C = 2

N % 2 == 0

N > 0

C = C * C
N = N - 1

N % 2 == 0

C = 0

print(C)
Control Flow Graph – Flattening

- Can we do better (i.e. destroy the graph)?
- Yes! We can flatten the graph
  - Technique called Chenxification after Chenxi Wang
  - Improved by Lazlo & Kiss
- The idea:
  - Replace the whole program by a big switch / case
  - Put all instruction blocks in it
  - Jump on blocks depending on a control value
Control Flow Graph - Example

N = RAND()
C = 2

N % 2 == 0

N > 0

C = C * C
N = N - 1

print(C)

C = 0

Switch(ctrl)

ctrl = 0

N = RAND()
C = 2
ctrl = 1

C = 0
ctrl = 4

C = C * C
N = N - 1

ctrl = 2

print(C)

ctrl = 123

N%2=0 ? ctrl = 2 : ctrl = 3

N > 0 ? ctrl = 5 : ctrl = 4

print(C)

ctrl = 123

C = 0
ctrl = 4
Control Flow Graph - Example

1. \( N = \text{RAND()} \)
2. \( C = 2 \)
3. \( N \equiv 0 \) → \( C = 0 \)
4. \( N > 0 \) → \( C = C \times C \)
5. \( N = N - 1 \)
6. \( C = 0 \)
7. \( \text{print}(C) \)

Switch based on 'ctrl':
- \( \text{if } N \% 2 = 0 \) then \( \text{ctrl} = 2 \) else \( \text{ctrl} = 3 \)
- \( \text{if } N > 0 \) then \( \text{ctrl} = 5 \) else \( \text{ctrl} = 4 \)
- Print \( C \)
- \( \text{ctrl} = 123 \)
- \( \text{ctrl} = 4 \)
Control Flow Graph - Example

```
ctrl = 0
Switch(ctrl)

N = RAND()
C = 2
ctrl = 1

C = C * C
N = N - 1
ctrl = 2
print(C)

ctrl = 123
N % 2 == 0 ? ctrl = 2 : ctrl = 3

0

1

2

N > 0 ? ctrl = 5 : ctrl = 4

3

4

C = 0
ctrl = 4

ctrl = 123
```
Control Flow Graph - Example

\begin{itemize}
  \item \texttt{ctrl = 0}
  \item Switch(\texttt{ctrl})
  \item \texttt{N = RAND()}
  \item \texttt{C = 2}
  \item \texttt{ctrl = 1}
  \item \texttt{N \% 2 = 0 \? \texttt{ctrl = 2 : ctrl = 3}}
  \item \texttt{N > 0 \? \texttt{ctrl = 5 : ctrl = 4}}
  \item print(\texttt{C})
  \item \texttt{ctrl = 123}
  \item \texttt{C = C \* C}
  \item \texttt{N = N - 1}
  \item \texttt{ctrl = 2}
  \item \texttt{C = 0}
  \item \texttt{ctrl = 4}
  \item \texttt{END}
\end{itemize}
```python
import struct
import binascii
import math

lrot = lambda x, n: (x << n) | (x >> (32 - n))

class MD5:
    A, B, C, D = (0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476)

    # r specifies the per-round shift amounts
    r = [7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21]

    # Use binary integer part of the sines of integers (Radians) as constants
    k = [int(math.floor(abs(math.sin(i + 1)) * (2 ** 32))) for i in range(64)]

    def _init_(self, message):
        length = struct.pack('<Q', len(message) * 8)
        while len(message) > 64:
            self._handle(message[:64])
            message = message[64:]
        message += '\x80'
        message += '\x00' * ((56 - len(message)) % 64) % 64
        message += length
        while len(message):
            self._handle(message[:64])
            message = message[64:]

    def _handle(self, chunk):
        w = list(struct.unpack('<' + 'I' * 16, chunk))

        a, b, c, d = self.A, self.B, self.C, self.D
        for i in range(64):
            if i < 16:
                f = (b & c) | ((~b) & d)
                g = i
            elif i < 32:
                f = (d & b) | ((~d) & c)
                g = (5 * i + 1) % 16
            elif i < 48:
                f = b & c | (d & ~b)
                g = (3 * i + 5) % 16
            else:
                f = c & (b | (~d))
                g = (7 * i + 1) % 16
            A = D + E + (W[i] + f) + k[i] + S16[i]
            E = D
            D = C
            C = B
            B = A
            W[i] = W[i] + g
```

---

**Before**

```python
import struct
import binascii
import math

lrot = lambda x, n: (x << n) | (x >> (32 - n))

class MD5:
    A, B, C, D = (0x67452301, 0xefcdab89, 0x98badcfe, 0x10325476)

    # r specifies the per-round shift amounts
    r = [7, 12, 17, 22, 7, 12, 17, 22, 7, 12, 17, 22, 5, 9, 14, 20, 5, 9, 14, 20, 5, 9, 14, 20, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 4, 11, 16, 23, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21, 6, 10, 15, 21]

    # Use binary integer part of the sines of integers (Radians) as constants
    k = [int(math.floor(abs(math.sin(i + 1)) * (2 ** 32))) for i in range(64)]

    def _init_(self, message):
        length = struct.pack('<Q', len(message) * 8)
        while len(message) > 64:
            self._handle(message[:64])
            message = message[64:]
        message += '\x80'
        message += '\x00' * ((56 - len(message)) % 64) % 64
        message += length
        while len(message):
            self._handle(message[:64])
            message = message[64:]

    def _handle(self, chunk):
        w = list(struct.unpack('<' + 'I' * 16, chunk))

        a, b, c, d = self.A, self.B, self.C, self.D
        for i in range(64):
            if i < 16:
                f = (b & c) | ((~b) & d)
                g = i
            elif i < 32:
                f = (d & b) | ((~d) & c)
                g = (5 * i + 1) % 16
            elif i < 48:
                f = b & c | (d & ~b)
                g = (3 * i + 5) % 16
            else:
                f = c & (b | (~d))
                g = (7 * i + 1) % 16
            A = D + E + (W[i] + f) + k[i] + S16[i]
            E = D
            D = C
            C = B
            B = A
            W[i] = W[i] + g
```
Putting it all together

- We have three obfuscation transforms
- We should be able to combine them
  - Choose the correct order to maximize efficiency
  - Use data obfuscation to mask flattening control
  - Optionally iterate some transforms
- Keep in mind the performance impact
  - The execution time can increase significantly
  - The program size can explode
  - Maybe necessary to target sensitive functions
Putting it all together

- Keep in mind the performance loss

<table>
<thead>
<tr>
<th></th>
<th>SIZE</th>
<th>TIME</th>
</tr>
</thead>
<tbody>
<tr>
<td>FLATTENING</td>
<td>+ 100 %</td>
<td>&lt; +10 %</td>
</tr>
<tr>
<td>RENAMING</td>
<td>+0 %</td>
<td>+0 %</td>
</tr>
<tr>
<td>ARRAY ALIASING</td>
<td>x 10</td>
<td>+11 %</td>
</tr>
</tbody>
</table>
Before

```
import struct

def _init_(self, message):
    length = struct.pack('<Q', len(message) * 8)
    while len(message) > 64:
        self._handle(message[:64])
        message = message[64:]

    message += '\x00' * (56 - len(message) % 64)
    message += length
    while len(message):
        self._handle(message[:64])
        message = message[64:]

    def _handle(self, chunk):
        w = list(struct.unpack('< ' + 'I' * 16, chunk))

        a, b, c, d = self.A, self.B, self.C, self.D

        for i in range(64):
            if i < 16:
                f = (b & c) | ((~b) & d)
                g = i
            elif i < 32:
                f = (d & b) | ((~d) & c)
                g = (5 * i + 1) % 16
            elif i < 48:
                f = b ^ c & d
                g = (3 * i + 5) % 16
            else:
                f = c ^ (b | (~d))
                g = (7 * i) % 16

            x = b + 1rot((a + f + self.k[i] + w[g]) &
            a, b, c, d, = d, d, x & 0xffffffff, b, c

        self.A = (self.A + a) & 0xffffffff
        self.B = (self.B + b) & 0xffffffff
        self.C = (self.C + c) & 0xffffffff
        self.D = (self.D + d) & 0xffffffff
```

After

```
import struct

def _init_(self, _39d98514e2):
    _907be1bf9 = _050dada3a32[4]
    _050dada3a32[4] = _907be1bf9


    w = list(struct.unpack('< ' + 'I' * 16, chunk))

    a, b, c, d = self.A, self.B, self.C, self.D

    for i in range(64):
        if i < 16:
            f = (b & c) | ((~b) & d)
            g = i
        elif i < 32:
            f = (d & b) | ((~d) & c)
            g = (5 * i + 1) % 16
        elif i < 48:
            f = b ^ c & d
            g = (3 * i + 5) % 16
        else:
            f = c ^ (b | (~d))
            g = (7 * i) % 16

        x = b + 1rot((a + f + self.k[i] + w[g]) &
        a, b, c, d, = d, d, x & 0xffffffff, b, c

        self.A = (self.A + a) & 0xffffffff
        self.B = (self.B + b) & 0xffffffff
        self.C = (self.C + c) & 0xffffffff
        self.D = (self.D + d) & 0xffffffff
```
Conclusion

- We achieve a nice looking obfuscation
  - Using somehow simple transforms
- But might not hold against advanced analysis
  - In particular dynamic analysis
    → Debugging, Symbolic execution, etc

- What about dynamic obfuscation?
  → Self modifying programs, white box crypto, etc
Conclusion

- We achieve a nice looking obfuscation
  - Using somehow simple transforms
- But might not hold against advanced analysis
  - In particular dynamic analysis
    → Debugging, Symbolic execution
- What about dynamic obfuscation?
  → Self modifying programs, white box crypto, etc
ANY QUESTIONS?

Thank you for your attention