Obfuscation by Interpretation

Jean-Yves Marion
Daniel Reynaud, PhD Student
reynaudd@loria.fr

Definitions

Low-level language :

An executable language with the following properties :
(P1) - code overwriting is allowed
(P2) - code and data can be mixed
(P3) - the control can be transferred anywhere (arbitrary jumps)
(ex: Assembly languages)

High-level language :

A language with low-level safety properties (ie without P1, P2 and P3)
(ex: Javascript, Java bytecode, .NET bytecode...)

Plan

1. Obfuscation of high-level languages with low-level properties

2. Examples

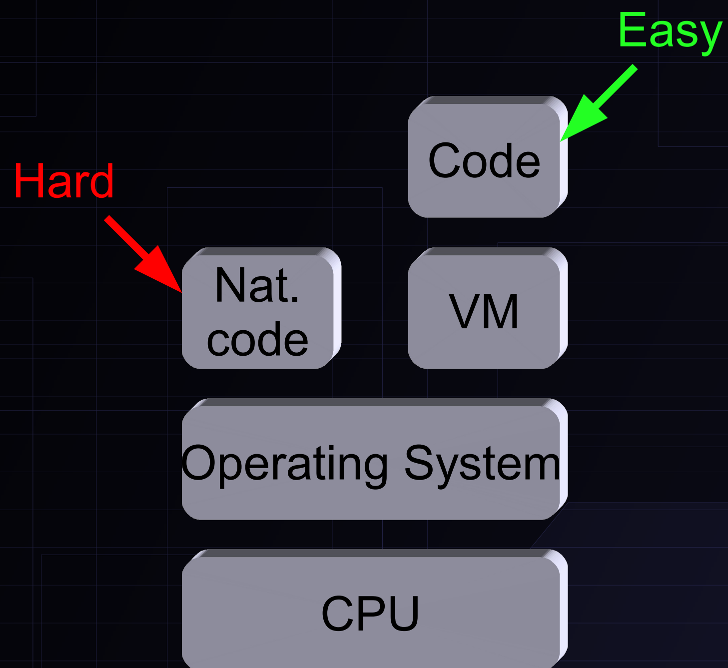
3. Formal approach

Conclusion

Jean Yves Marion, Daniel Reynaud reynaudd@loria.fr

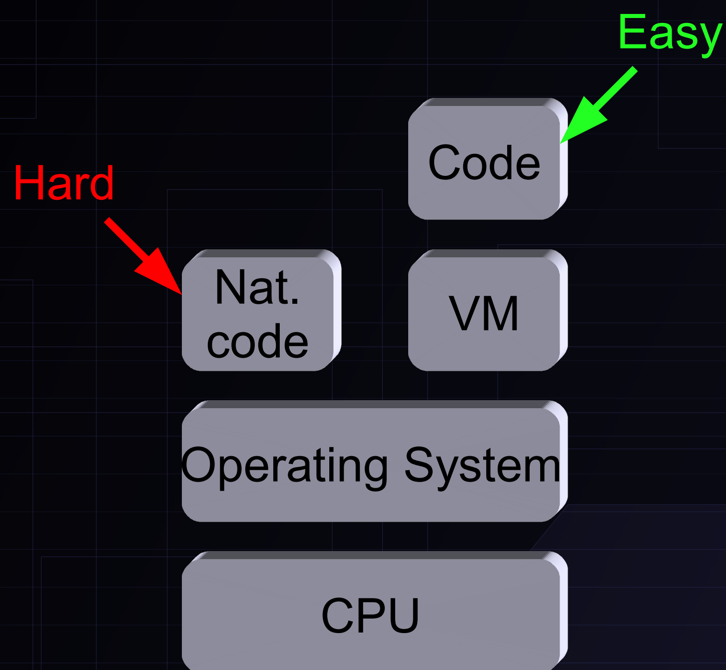
Reverse Engineering:

is usually seen like this:

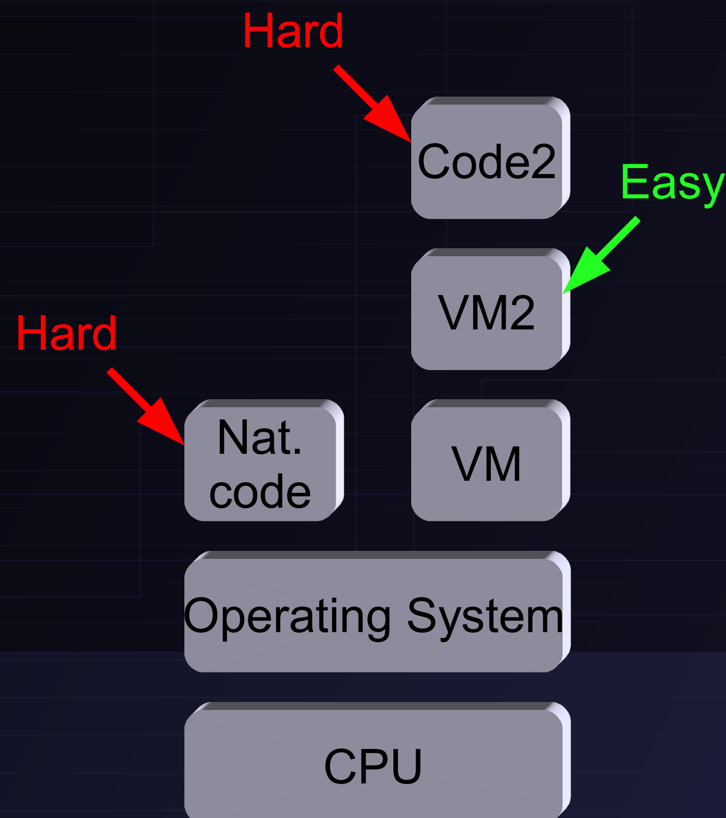


Reverse Engineering:

is usually seen like this:



but if an interpretation layer is added:



Pros

- the interpreted code is as hard to reverse engineer as a native application (virtually all obfuscations for native programs work in the custom language)

- obfuscating is easy, it consists in the compilation from the high-level language to the custom low-level language

- every acceptable language allows an interpreter for a low-level language to be programmed

Cons

- linear slowdown of the original program

Plan

1. *Obfuscation of high-level languages with low-level properties*
2. *Examples*
3. *Formal approach*

Conclusion

Jean Yves Marion, Daniel Reynaud reynaudd@loria.fr

Languages used:

- the chosen high-level language is **Javascript**

- the chosen low-level language is a **« pseudo-assembly »** (but any language with properties P1, P2 and P3 could act as an « obfuscated language »)

Pseudo Assembly:

- arithmetic and logic instructions : ADD, SUB, MUL, DIV, OR, XOR, NOT, AND

- data transfer instructions : MOV, PUSH, POP

- control transfer instructions : JMP, JE, JG, CALL, RET, SYSCALL

- the virtual CPU uses some classic registers (EAX, EBX, ECX, EDX, EIP, ESP...) and a flat memory model

Hello World:

```
pushs [@hello]
push 1
pushs [@alert]
syscall
exit 0

hello:
db '"HelloWorld"'

alert:
db 'alert'
```

Hello World Again:

```
jmp [@main]

hello:
db '"HelloWorld"'
alert:
db 'alert'

main:
pushs [@hello]
push 1
pushs [@alert]
syscall
exit 0
```

Code Overwriting (P1):

```
call [@overwrite]
pushs [@hello]
push 1
pushs [@alert]
syscall
exit 0

overwrite:
mov [@alert] 0x7072696e
ret

hello:
db '"HelloWorld"'
alert:
db 'alert'
```

Code and Data Mixing (P2) and Arbitrary Jumps (P3):

```
jmp @skip

db 0x0250

skip:
jmp -2
```

XOR Encryption/Decryption Loop:

```
call [@xorize]
jmp [@main]

hello:
db '"HelloWorld"'
alert:
db 'alert'

xorize:
mov eax 16
mov ebx @hello
add ebx eax
```

```
  loop:
sub eax 4
sub ebx 4
xor [ebx] 0x42424242
cmp eax 0
jne [@loop]
ret

main:
pushs [@hello]
push 1
pushs [@alert]
syscall
exit 0
```

Plan

1. *Obfuscation of high-level languages with low-level properties*

2. *Examples*

3. *Formal approach*

Conclusion

Jean Yves Marion, Daniel Reynaud reynaudd@loria.fr

Notations:

L refers to our high-level programming language
M refers to our low-level programming language

f is a compiler from L to M (in our setting : an obfuscator)
g is a compiler from M to L (in our setting : a deobfuscator)

[x] is the function computed by program x
[x](y) is the output of the execution of program x on input y

S is a specialisation program, written in L
Int is an interpreter for M programs, written in L

Rogers' Isomorphism Theorem:

Between any two programming languages L and M defining acceptable enumerations of the partial recursive functions on natural integers, there exists a compiling bijection.

In our setting we can suppose we have the compiling function [f] from L to M (one-to-one and onto) and build the inverse function [g] :

```
g(p1) {
    p = 0;
    while(true) {
        if(f(p) == p1)
            return p;
        p++;
    }
}
```

g exists, it is total and computable but it is not an efficient deobfuscator due to its time-complexity.

Relaxing the Requirements for the Deobfuscator:

- [f] (resp. [g]), instead of a bijection (resp. its inverse function) can be viewed as a simple compiling function from L to M (resp. M to L) without loss of generality

- as a result, the composition of [f] and [g] is not the identity function but produces functionally equivalent programs

- by using the second Futamura projection, we obtain a generic compiler from M to L : g = [S](S, int)

- g might or might not be a good deobfuscator depending on the implementation of S and on the properties of L and M

- informally : constraining the reverse engineer to study S(int, CODE) is achieving a « weak virtual black-box property »

Future Work:

- compilation from L to M (i.e. from high-level to low-level) is easy. Decompilation from M to L is intuitively hard, is it provably hard ?

- the hardness comes from the fact that L and M have different properties P1, P2 and P3. Is it possible to introduce new properties Pn such that compiling from M to L is harder or provably hard ?

- how to define a criterion for a « good deobfuscator » ? (i.e. a compiler from M to L)

- according to this criterion, are there better deobfuscators than the generic deobfuscators discussed earlier ?

Plan

1. *Obfuscation of high-level languages with low-level properties*
2. *Examples*
3. *Formal approach*

Conclusion

∀

Conclusion:

- proofs of complexity have yet to come

- every acceptable language (especially, high-level languages) allow to express interpreters for low-level languages. This means that low-level obfuscations can be applied to high-level programs compiled and then interpreted.

- therefore, the « high-levelness » of a language is not enough to ensure easy reverse engineering.