Obfuscator-LLVM

Software Obfuscation for the Masses

area41 // Zürich // 02-06-2014
Who?
Who?

- Pascal Junod (@cryptopathe), Julien Rinaldini (@pyknite), Johan Wehrli (@jowehrli) // HEIG-VD
- Julie Michielin // Kudelski Security
- Several bachelor & master students
Why?
Black-Box Adversaries

- Play the security game according to the rules
- Interact with components according to the defined APIs
- Adversaries considered in most « provably-secure » schemes by cryptographers
Grey-Box Adversaries

• Adversaries looking to exploit additional « side-channel » information
  • Time
  • Power/EM leakage
  • Faults
White-Box Adversaries

- (Almost) completely master SW/HW
- Can read every memory
- Can disturb every computation at will
White-Box Adversaries

- Examples in real life:
  - DRM circumventing
  - License management scheme cracking
  - Rogue SW reverse engineering, IP stealing
  - Malware analysis
  - Interoperability work
  - Security audits
Attack Scenarios

- Reverse engineering
- IP stealing, cryptographic secret extractions
- Code modification
- Code distribution
Main Motivations

- Bad guys use software protection all the time. Why not good guys?
- Performing research on automatic desobfuscation not always easy (costly commercial tools)
- No open-source C/C++ obfuscation tool available that is capable of performing code transformation
Problems

• Dual-use tools

• Example of the iOS 7 jailbreak Christmas drama
  • Obfuscated code
    • 0-day jailbreaking exploit?
  • Bundled pirate app repository?
  • See http://crypto.junod.info/2013/12/24/about-obfuscator-llvm-dual-use-tools-and-academic-ethics/

• How to audit a SW protection tool with respect to backdoor insertion?
What?
Software Protection

- Goals of software protection:
  - Code more difficult to understand
  - Code more difficult to modify
  - Code more diverse

Obfuscation

Tamperproofing

Watermarking
@P=split//,".URRUU\c8R";@d=split//,"\nrekcah
xinU / lreP rehtona tsuJ";sub p{
    @p{"r$p","u$p"}=(P,P);pipe "r$p","u$p"; ++$p;
    ($q*=2) ++$f=!fork;map{ $P=$P[$f^ord ($p{$_})&6]; $p{$_}/ ^$P/ix?$P: close$_ }keys %p}p;p;p;p;p;
    map{ $p{$_}/ ^$P/ext?$P:close$_ } keys %p;p;p;p;p;
    map{ $p{$_}/ ^P/&&<$_> } %p;
    $_=$d[$q];sleep rand(2)if/\S/;print
for(v A((u A((e A((r-2?0:(V A(1[U]))),"C")
),system("stty raw -echo min 0"),fread(l,78114,1,e),B(e),"B")),"A")); 118- (x =*c++);
(y=x/8%8,z=(x&199)-4 S 1 S 1 S 186 S 2 S 2 S 3 S 0, r=(y>5)*2+y, z=(x& 207)-1 S 2 S 6 S 2 S 182 S 4)?D(0)D(1)D(2)D(3)D(4)D(5)D(6)D(7)(z=x-2 C C C C C C
C C C+C+129 S 6 S 4 S 6 S 8 S 8 S 6 S 2 S 2 S 2 S 12)?x/64-1?((0 O a(y)=a(x) O 9
[o]=a(5),8[o]=a(4) O 237==*c++?((int (*)())(2-*c++?fwrite:fread))(1+k1[k]*
256,128,1,(fseek(y=5[k]-1?u:v,((3[k]|4[k]<<8)<<7|2[k])<<7,Q=0),y))=0 O y=a(5),
z=a(4),a(5)=a(3),a(4)=a(2),a(3)=y,a(2)=z O c=1+d(5) O y=1[x=d(9)],z=l[++x],
x[1]=a(4),l[++]=a(5),a(5)=y,a(4)=z O 2-*c?Z | read(0,&Z,1),1&c++?Q=Z,Z=0:
(Q=!Z):(c++,Q=r=V?fgetc(V):-1,s=s&~1|r<0) O++c,write(1, &7[o], 1) O z=c+2-1,w,
c=1+q O p,c=1+z O c=1+q O s'=1 O Q=q[1] O s=1 O q[1]=Q O Q=~Q O a(5)=l[x=q],
a(4)=l[++x] O s=|s&16|9<Q%16?Q+=6,16:0,z|s=|1&s|Q>159?Q+=96, 1:0,y=Q,h(s<<8)
O 1[x=q]=a(5),l[++]=a(4) O x=Q%2,Q=Q/2+s%2,s=|1 &s|Q>>7 O 1[d(3)]=Q O m y n(0,-7)y) O m z=0,y=|x=h(y) O m z=0,
y=Q=x,h(y) O m z=Q/2|2*x,y=Q=x,h(y) O m Q n(s%2,-7)y) O m Q n(0,-7)y) O m Q n(s%2,+
7)y) O m Q n(0,+,7)y) O z=r-8?d(r+1):s|Q<<8,w O p,r-8?o[r+1]=z,r
[o]=z>>8:(s=-40&z|2,Q=z>>8) O r[o]--|--o[r-1]O a(5)=z=a(5)+r[o],a(4)=z=a(4)+
or[r-1]+z/256,s=|1&s|z>&8 0 ++o[r+1] | r[o]++o [r+1]=*c++,r[o]=*c++0 z=c-1,w,
c=y*8+1 O x=q,b z=c-1,w,c=1+x) O x=q,b c=1+x) O b p,c=1+z) O a(y)=*c++0 r=y
,x=0,a(r)n(1,+,y)s<<8) O r=y,x=0,a(r)n(1,+,y)s<<8))))

system("stty cooked echo"); B((B((V?B(V):0,u)),v)); }
On the (Im)possibility of Obfuscating Programs

Boaz Barak† Oded Goldreich‡ Russell Impagliazzo§ Steven Rudich†
Amit Sahai‖ Salil Vadhan** Ke Yang††

July 29, 2010

Abstract

Informally, an obfuscator $O$ is an (efficient, probabilistic) “compiler” that takes as input a program (or circuit) $P$ and produces a new program $O(P)$ that has the same functionality as $P$ yet is “unintelligible” in some sense. Obfuscators, if they exist, would have a wide variety of cryptographic and complexity-theoretic applications, ranging from software protection to homomorphic encryption to complexity-theoretic analogues of Rice’s theorem. Most of these applications are based on an interpretation of the “unintelligibility” condition in obfuscation as meaning that $O(P)$ is a “virtual black box,” in the sense that anything one can efficiently compute given $O(P)$, one could also efficiently compute given oracle access to $P$.

In this work, we initiate a theoretical investigation of obfuscation. Our main result is that, even under very weak formalizations of the above intuition, obfuscation is impossible. We prove this by constructing a family of efficient programs $P$ that are unobfuscatable in the sense that (a) given any efficient program $P'$ that computes the same function as a program $P \in P$, the “source code” $P$ can be efficiently reconstructed, yet (b) given oracle access to a (randomly selected) program $P \in P$, no efficient algorithm can reconstruct $P$ (or even distinguish a certain bit in the code from random) except with negligible probability.

We extend our impossibility result in a number of ways, including even obfuscators that (a) are not necessarily computable in polynomial time, (b) only approximately preserve the functionality, and (c) only need to work for very restricted models of computation ($TC^0$). We also rule out several potential applications of obfuscators, by constructing “unobfuscatable” signature schemes, encryption schemes, and pseudorandom function families.
Caveat Emptor

- Source vs. binary obfuscation
- Supported languages/platforms
- Associated cost
- Resistance
Well-Known Techniques

- Packing
- Anti-debugging tricks insertion
- Code interleaving
- Code transformation
- Code virtualization
- ...

How?
Home

cryptopathè edited this page on 20 mai · 23 revisions

Obfuscator-LLVM is a project initiated in June 2010 by the information security group of the University of Applied Sciences and Arts Western Switzerland of Yverdon-les-Bains (HEIG-VD).

The aim of this project is to provide an open-source fork of the LLVM compilation suite able to provide increased software security through code obfuscation and tamper-proofing. As we currently mostly work at the Intermediate Representation (IR) level, our tool is compatible with all programming languages (C, C++, Objective-C, Ada and Fortran) and target platforms (x86, x86-64, PowerPC, PowerPC-64, ARM, Thumb, SPARC, Alpha, Cell/SPU, MIPS, MSP430, SystemZ, and XCore) currently supported by LLVM.
LLVM

• Complete compilation framework, competitor of GCC

• Project supported by Apple since 2005

• Very dynamic community, state-of-the-art SW architecture

• Front-ends available for C, C++, Objective-C, Fortran, Ada, Haskell, Python, Ruby, ...

• Back-ends available for x86, x86-64, PowerPC, PowerPC-64, ARM, Thumb, Sparc, Alpha, MIPS, MSP430, SystemZ, XCore
Instructions Substitution
Instructions Substitution

• Replace an arithmetic or Boolean expression by an equivalent one

  ● $A \land B = (A \& \neg B) \lor (\neg A \& B)$
  ● $A + B = A - (\neg B)$
  ● $A+B = (A+R) + (B+R) - 2*R$
  ● ...
Example: AES Implementation

-mllvm -sub
-mllvm -sub-loop=3
Bogus Control Flow Insertion
-mllvm -bfc
Control Flow Flattening
-mllvm -fla
import Foundation

let interestingNumbers = [
    "Prime": [2, 3, 5, 7, 11, 13],
    "Fibonacci": [1, 1, 2, 3, 5, 8],
    "Square": [1, 4, 9, 16, 25],
]

var largest = 0
for (kind, numbers) in interestingNumbers {
    for number in numbers {
        if number > largest {
            largest = number
        }
    }
}

println("Hello, World!")
println(largest)
@Tv4main18interestingNumbersGVSs10DictionarySGSSaSi__ = global %VSs10Dictionary
zeroinitializer, align 8
@Tv4main7largestSi = global %Si zeroinitializer, align 8
"_swift_FORCE_LOAD_$._swiftFoundation" = external global i1
"_swift_FORCE_LOAD_$._swiftFoundation_$._main" = weak hidden constant i1*
"_swift_FORCE_LOAD_$._swiftFoundation"
"_swift_FORCE_LOAD_$._swiftDarwin" = external global i1
"_swift_FORCE_LOAD_$._swiftDarwin_$._main" = weak hidden constant i1*
"_swift_FORCE_LOAD_$._swiftDarwin"
"_swift_FORCE_LOAD_$._swiftObjectiveC" = external global i1
"_swift_FORCE_LOAD_$._swiftObjectiveC_$._main" = weak hidden constant i1*
"_swift_FORCE_LOAD_$._swiftObjectiveC"
"_swift_FORCE_LOAD_$._swiftDispatch" = external global i1
"_swift_FORCE_LOAD_$._swiftDispatch_$._main" = weak hidden constant i1*
"_swift_FORCE_LOAD_$._swiftDispatch"
"_swift_FORCE_LOAD_$._swiftCoreGraphics" = external global i1
"_swift_FORCE_LOAD_$._swiftCoreGraphics_$._main" = weak hidden constant i1*
"_swift_FORCE_LOAD_$._swiftCoreGraphics"
0 = private unnamed_addr constant [6 x i16] [i16 80, i16 114, i16 105, i16 109, i16 101, i16 0]
@metadata = internal constant %swift.full_heapmetadata { void (%swift.refcounted **)
* @arraydestroy, i8** null, %swift.type { i64 65 } }
_TMdSi = external global %swift.full_type
Command Line Magics

• pjunod$ /Applications/Xcode6-Beta.app/Contents.//Developer/Toolchains/XcodeDefault.xctoolchain/usr/bin/swift -sdk /Applications/Xcode6-Beta.app/Contents/Developer/Platforms/MacOSX.platform/Developer/SDKs/MacOSX10.10.sdk/ -emit-ir -o main.ir main.swift

• pjunod$ /Volumes/scratch/devel/build/bin/opt main.ir -o main_fla.ir -S -std-compile-opts -fla -O1
Code Flattening

- Routing variable generation
  - Weak: hard-coded
  - Better: dynamically generated using opaque constructs
  - Even better: depending on the inner state of the program (tamper-proofing)
- Coming soon: basic-block splitting before flattening
Procedures Merging
Procedure Merging

- In a compilation unit, put the code of all routines in a single one (that can be later flattened, etc.)
- Use the initial symbol as a wrapper to the huge routine
  - Responsible to handle parameters
- Reverse engineer has to figure out the signature of each function
- Not useful for exported APIs
Current State of O-LLVM

- Published
  - Instruction substitutions
  - Bogus-control flow insertion
  - Basic code flattening
Current State of O-LLVM

- In testing phase
- Procedure merging
- Basic-block splitting
- More resistant code flattening
- Developer annotations
Current State of O-LLVM

• In implementation phase

• Code tamper proofing (post-processor has to be re-written)
Current State of O-LLVM

- Foreseen / wished
- Anti-debugging tricks insertion
- Packing
- Code virtualization
- ???
Questions?

http://o-llvm.org
@ollvm