Binary Obfuscation from the Top Down

How to make your compiler do your dirty work.

Binary Obfuscation

Why Top Down?

- Assembly, while “simple,” is tedious.
- It’s easier for us to write higher-level code.
  - Some of us.
- Why do it by hand when you can be lazy?

Binary Obfuscation

What's the purpose of obfuscation?

- To waste time.
- To intimidate.
- To be a total jerk.

Binary Obfuscation

What tools will be used?

- C and C++
- MSVC++ for compilation (sorry)

Binary Obfuscation

What will not be covered?

- Anti-debug
- Source obfuscation where it does not relate to binary transformations
- Obfuscation effectiveness
- Post-compilation obfuscation

Important Basics

Hopefully we can get through this *really quickly*.

Fun With Pointers

car cdr cadr cdar cdadr cdddr caar caaar caaaar caaaaaar

Binary Obfuscation

Function Pointers

- Like string-format vulnerabilities, function pointers are ancient Voodoo.
- I honestly don't know who thought these were a good idea, but I freakin' love 'em.
- See src/funcptr.c

Binary Obfuscation

Function Pointers

```
int foo (void) {
  return 949;
}

int bar (void) {
  int (*fooPtr)(void);
  fooPtr = foo;
  return fooPtr();
}
```

Binary Obfuscation

Method Pointers

- Abuse of method pointers would probably make Bjarne Stroustrup really angry.
- There is also one thing uglier than function pointers. That's method pointers.
- See src/methodptr.cpp

Binary Obfuscation

Method Pointers

```
int MyClass::foo(void) {
  return 310;
}

int bar (void) {
  MyClass baz;
  int (MyClass::*fooPtr)(void);
  fooPtr = &MyClass::foo;
  return (MyClass.*baz)fooPtr();
}
```

Calling Conventions

I really want to write a clever pun about payphones and DEFCON, but I just can't.

Binary Obfuscation

Calling Conventions

- When making a function call, there are a few ways to do it:
  - stdcall
  - cdecl
  - fastcall
  - thiscall

Binary Obfuscation

Calling Conventions

- stdcall
  - Push arguments onto stack
  - Called function pops from stack
  - Cleans up its own mess.

Binary Obfuscation

Calling Conventions

- cdecl
  - Push arguments onto stack
  - Called function pops from stack
  - Called function cleans up the mess

Binary Obfuscation

Calling Conventions

- fastcall
  - First two arguments less than a DWORD moved into ecx and edx respectively
  - Rest are pushed onto the stack
  - Called function pops from the stack
  - Called function cleans up the mess

Binary Obfuscation

Calling Conventions

- thiscall
  - Used when a function within a class object is called
  - “this” pointer moved into ecx
  - Function arguments pushed onto stack
  - Called function pops from stack
  - Cleans up its own mess

Compiler Optimizations

The Dragon Book: Not Just for Furries Anymore

Binary Obfuscation

Compiler Optimizations

- Control-flow analysis
- Variable analysis
- Reach-of-use
- The `volatile` keyword

Binary Obfuscation

Compiler Optimizations

- At compile time, your code is separated into multiple blocks.
- A “block” consists of code separated by conditional (e.g. JLE, JNE, etc.) and unconditional jumps (e.g. CALL and JMP).
- How this code is organized and how the jumps occur affects the optimization of the program.

Binary Obfuscation

Compiler Optimizations

```
MOV EAX,949
XOR EAX,310
CMP EAX,0
JNE z0r
```

```
z0r:
XOR EAX,310
PUSH EAX
```

```
XOR EAX,949
LEAVE
RETN
```

Binary Obfuscation

Compiler Optimizations

```
MOV EAX,949
XOR EAX,310
CMP EAX,0
JNE z0r
```

lol lemme fix this

```
z0r:
XOR EAX,310
PUSH EAX
```

```
XOR EAX,949
LEAVE
RETN
```

Binary Obfuscation

Compiler Optimizations

```
MOV EAX,949
XOR EAX,310
XOR EAX,310
PUSH EAX
```

Binary Obfuscation

Compiler Optimizations

- The compiler also looks at your variables to make sure you're not doing anything repetitive or inconsequential.
- Algorithms like the directed acyclic graph (DAG) algorithm and static variable analysis make sure memory and math are fully optimized.

Binary Obfuscation

Compiler Optimizations

```
MOV EAX,949
XOR EAX,310
XOR EAX,310
PUSH EAX
```

Binary Obfuscation

Compiler Optimizations

```
MOV EAX,949
XOR EAX,310
XOR EAX,310
PUSH EAX
```

lol seriously?

Binary Obfuscation

Compiler Optimizations

```
MOV EAX,949
PUSH EAX
```

Binary Obfuscation

Compiler Optimizations

```
MOV EAX,949
XOR EAX,310
CMP EAX,0
JNE z0r
```

```
z0r:
XOR EAX,310
PUSH EAX
```

```
XOR EAX,949
LEAVE
RETN
```

```
MOV EAX,949
PUSH EAX
```

Binary Obfuscation

Compiler Optimizations

- Your compiler is a neat-freak.
- If the compiler notices it doesn't need a variable anymore, it's just going to get rid of it, no matter what else you do to it.

Binary Obfuscation

Compiler Optimizations

```
MOV EAX,949
MOV EBX,310
MOV ECX,213
XOR EAX,EBX
ADD EBX,EAX
SUB EAX,EAX
PUSH EBX
PUSH EAX
```

Binary Obfuscation

Compiler Optimizations

```
MOV EAX,949
MOV EBX,310
MOV ECX,213
XOR EAX,EBX
ADD EBX,EAX
SUB EAX,EAX
PUSH EBX
PUSH EAX
```

Binary Obfuscation

Compiler Optimizations

```
MOV EAX,949
MOV EBX,310
XOR EAX,EBX
ADD EBX,EAX
SUB EAX,EAX
PUSH EBX
PUSH EAX
```

Binary Obfuscation

Compiler Optimizations

- There exist cases (mostly in hardware development) where you do NOT want your compiler to optimize your variable.
- This is where the `volatile` keyword comes in.
- Making your variable `volatile` tells the compiler not to do any optimizations to it.

Binary Obfuscation

Compiler Optimizations

```
volatile int foo;
volatile char bar;
volatile uint32_t baz;
```

Binary Obfuscation

Compiler Optimizations

```
int x;
x = 7;
x <<= 2;
x *= 2;
x -= 12;
x += (x*x)<<2;
printf("%d\n", x);
```

Binary Obfuscation

Compiler Optimizations

```
int x;
x = 7;
x <<= 2;
x *= 2;
x -= 12;
x += (x*x)<<2;
printf("%d\n", x);
```

→

```
PUSH 1E6C
PUSH “%d\n”
CALL $PRINTF
```

Binary Obfuscation

Compiler Optimizations

```
volatile int x;
x = 7;
x <<= 2;
x *= 2;
x -= 12;
x += (x*x)<<2;
printf("%d\n", x);
```

Binary Obfuscation

Compiler Optimizations

```
volatile int x;
x = 7;
x <<= 2;
x *= 2;
x -= 12;
x += (x*x)<<2;
printf("%d\n", x);
```

```
MOV [ESP],7
SHL [ESP],2
MOV EAX,[ESP]
ADD EAX,EAX
MOV [ESP],EAX
ADD [ESP],-0C
MOV ECX,[ESP]
MOV EDX,[ESP]
MOV EAX,[ESP]
IMUL ECX,EDX
...
```

Binary Formats

Everything is a file.

Binary Obfuscation

Binary Formats

- The most common formats you'll likely come across are the PE file format (Windows) and the ELF format (Linux).
- Both of these formats have a "table" they use for external library calls such as `printf`, `execv`, etc.
- For Windows it's called the IAT. For Linux it's the PLT.

Binary Obfuscation

Binary Formats

- If you obfuscate function pointers, they will likely not show up in those lists and therefore cause your library calls to fail.
- Circumventing this issue will be covered later.

Methods of Analysis

Know your opponent!

Binary Obfuscation

Methods of Analysis

- Someone can easily figure out the gist of what your program is doing by analyzing any of the API calls you make.
- There exist a few programs out there that already do this for you: VirusTotal and ZeroWine.

Binary Obfuscation

Methods of Analysis

- VirusTotal (virustotal.com) is a website that allows you to upload suspected malware files and analyze them against over thirty different scanners.
- At the end of the analysis is a list of all recognized Windows API calls made by the program, as well as various data sections within.

Binary Obfuscation

Methods of Analysis

- ZeroWine (zerowine.sourceforge.net) is a malware analysis tool that executes a program in a controlled environment and collects data.
- This, too, collects and reports on API calls made by the program, as well as any possible servers it may have contacted or files it may have written.

Binary Obfuscation

Methods of Analysis

- When analyzing a binary, there are two schools of analysis: live-code and dead-code.
- Dead-code is exactly how it sounds: you look at the binary, as-is, without executing.
- Live-code is the opposite: you run the program and watch what it does.

Binary Obfuscation

Methods of Analysis

- VirusTotal employs dead-code analysis. It simply reads the binaries uploaded to it, scans it with various virus scanners and reports.
- ZeroWine, however, employs live-code analysis. It runs the suspected program in a controlled environment and watches what happens.

Binary Obfuscation

Methods of Analysis

- Dead-code analysis can be frustrated through polymorphism.
- Live-code analysis can be frustrated through hiding, obfuscating and redirecting data and control-flow under the eyes of the reverser.

Obfuscation

We're almost at the fun part, I promise!

Binary Obfuscation

Obfuscation

- There are three separate classes of obfuscation.
  - Layout
  - Control-flow
  - Data

Binary Obfuscation

Obfuscation

- Layout obfuscation essentially means scrambling the program around at the source-level.
- The International Obfuscated C Contest (ioccc.org) is a perfect example of this.

Binary Obfuscation

Obfuscation

Anders Gavare, http://www0.us.ioccc.org/2004/gavare.c

```
X=1024; Y=768; A=3;

J=0;K=-10;L=-7;M=1296;N=36;O=255;P=9;_=1<<15;E;S;C;D;F(b){E="1""111886:6:??AAF"
"FHHMMOO55557799@@>>>BBBGGIIKK"[b]-64;C="C@=::C@@==@=:C@=:C@=:C5""31/513/5131/"
"31/531/53"[b ]-64;S=b<22?9:0;D=2;}I(x,Y,X){Y?(X^=Y,X*X>x?(X^=Y):0,  I (x,Y/2,X
)):(E=X);      }H(x){I(x,    _,0);}p;q(        c,x,y,z,k,l,m,a,          b){F(c
);x-=E*M     ;y-=S*M           ;z-=C*M        ;b=x*       x/M+        y*y/M+z
*z/M-D*D    *M;a=-x              *k/M    -y*l/M-z        *m/M;    p=((b=a*a/M-
b)>=0?(I    (b*M,_     ,0),b    =E,      a+(a>b      ?-b:b)):     -1.0);}Z;W;o
(c,x,y,     z,k,l,    m,a){Z=!    c?      -1:Z;c     <44?(q(c,x          ,y,z,k,
l,m,0,0     ),(p>      0&&c!=     a&&         (p<W         ||Z<0)          )?(W=
p,Z=c):     0,o(c+          1,    x,y,z,         k,l,          m,a)):0     ;}Q;T;
U;u;v;w    ;n(e,f,g,             h,i,j,d,a,    b,V){o(0      ,e,f,g,h,i,j,a);d>0
&&Z>=0? (e+=h*W/M,f+=i*W/M,g+=j*W/M,F(Z),u=e-E*M,v=f-S*M,w=g-C*M,b=(-2*u-2*v+w)
/3,H(u*u+v*v+w*w),b/=D,b*=b,b*=200,b/=(M*M),V=Z,E!=0?(u=-u*M/E,v=-v*M/E,w=-w*M/
E):0,E=(h*u+i*v+j*w)/M,h-=u*E/(M/2),i-=v*E/(M/2),j-=w*E/(M/2),n(e,f,g,h,i,j,d-1
,Z,0,0),Q/=2,T/=2,       U/=2,V=V<22?7:  (V<30?1:(V<38?2:(V<44?4:(V==44?6:3))))
,Q+=V&1?b:0,T                +=V&2?b        :0,U+=V    &4?b:0)     :(d==P?(g+=2
,j=g>0?g/8:g/     20):0,j    >0?(U=     j    *j/M,Q      =255-    250*U/M,T=255
-150*U/M,U=255    -100    *U/M):(U    =j*j      /M,U<M           /5?(Q=255-210*U
/M,T=255-435*U           /M,U=255    -720*       U/M):(U       -=M/5,Q=213-110*U
/M,T=168-113*U    /       M,U=111                -85*U/M)      ),d!=P?(Q/=2,T/=2
,U/=2):0);Q=Q<    0?0:       Q>O?     O:            Q;T=T<0?    0:T>O?O:T;U=U<0?0:
U>O?O:U;}R;G;B    ;t(x,y      ,a,     b){n(M*J+M    *40*(A*x   +a)/X/A-M*20,M*K,M
*L-M*30*(A*y+b)/Y/A+M*15,0,M,0,P,  -1,0,0);R+=Q    ;G+=T;B   +=U;++a<A?t(x,y,a,
b):(++b<A?t(x,y,0,b):0);}r(x,y){R=G=B=0;t(x,y,0,0);x<X?(printf("%c%c%c",R/A/A,G
/A/A,B/A/A),r(x+1,y)):0;}s(y){r(0,--y?s(y),y:y);}main(){printf("P6\n%i %i\n255"
"\n",X,Y);s(Y);}
```

Binary Obfuscation

Obfuscation

- Control-flow obfuscation involves twisting the typical downward-flow of a program to into spaghetti code.
- It has the added benefit of obfuscating source while simultaneously upsetting the normal flow a reverse-engineer is used to.

Binary Obfuscation

Obfuscation

- Data obfuscation involves masking whatever data you have in your program by any means.
- Strings, numbers, even functions within your program can be masked, obfuscated, interwoven or encrypted without hand-writing any assembly.

Obfuscation Techniques

Now the fun begins.

Binary Obfuscation

Obfuscation Techniques

- The goal is to obfuscate the binary without doing binary transformations.
- We know how the compiler optimizes, what it does to our data and how it stores some information important for programmatic logic.
- With this in mind, we can now leverage our code against the compiler.

Binary Obfuscation

Obfuscation Techniques

- Layout obfuscation is essentially useless.
- Renaming variables, removing whitespace and using `#define` routines for functions typically has very little impact on the underlying program.
- Sure you can do layout obfuscation on your code, and some of it MAY translate to obfuscated code, but the signal-to-noise ratio is much too low for to be useful.

Control-Flow Obfuscation

Turn that boring linear NOP sled into something worthy of *Raging Waters.*

Binary Obfuscation

Control-Flow Obfuscation

- With function pointers, method pointers, the `volatile` keyword and the `goto` keyword on our side, we can do some really fun stuff.

Binary Obfuscation

Control-Flow Obfuscation

- Opaque predicates are tautological IF statements.
- An opaque predicate cannot be optimized because the compiler cannot determine the outcome.
- You see this frequently in obfuscated JavaScript.

Binary Obfuscation

Control-Flow Obfuscation

```
int a=7,b=2,c=8,d=9;
if (a+b+c*d > 0)
{
    puts(“yes”);
    exit(0);
}
puts(“no”);
```

Binary Obfuscation

Control-Flow Obfuscation

```
int a=7,b=2,c=8,d=9;
if (a+b+c*d > 0)
{
   puts(“yes”);
   exit(0);
}
puts(“no”);
```

→

```
PUSH “yes”
CALL $PUTS
PUSH 0
CALL $EXIT
```

Binary Obfuscation

Control-Flow Obfuscation

```
int a,b,c,d;
srand(time(0));
a=rand()+1;b=rand()+1;
c=rand()+1;d=rand()+1;
if (a+b+c*d > 0)
{
   puts(“yes”);
   exit(0);
}
puts(“no”);
```

Binary Obfuscation

Control-Flow Obfuscation

```
int a,b,c,d;
srand(time(0));
a=rand()+1;b=rand()+1;
c=rand()+1;d=rand()+1;
if (a+b+c*d > 0)
{
   puts(“yes”);
   exit(0);
}
puts(“no”);
```

→

```
...
TEST EAX,EAX
JLE SHORT :NO
PUSH “yes”
CALL $PUTS
PUSH 0
CALL $EXIT
NO: PUSH “no”
CALL $PUTS
```

Binary Obfuscation

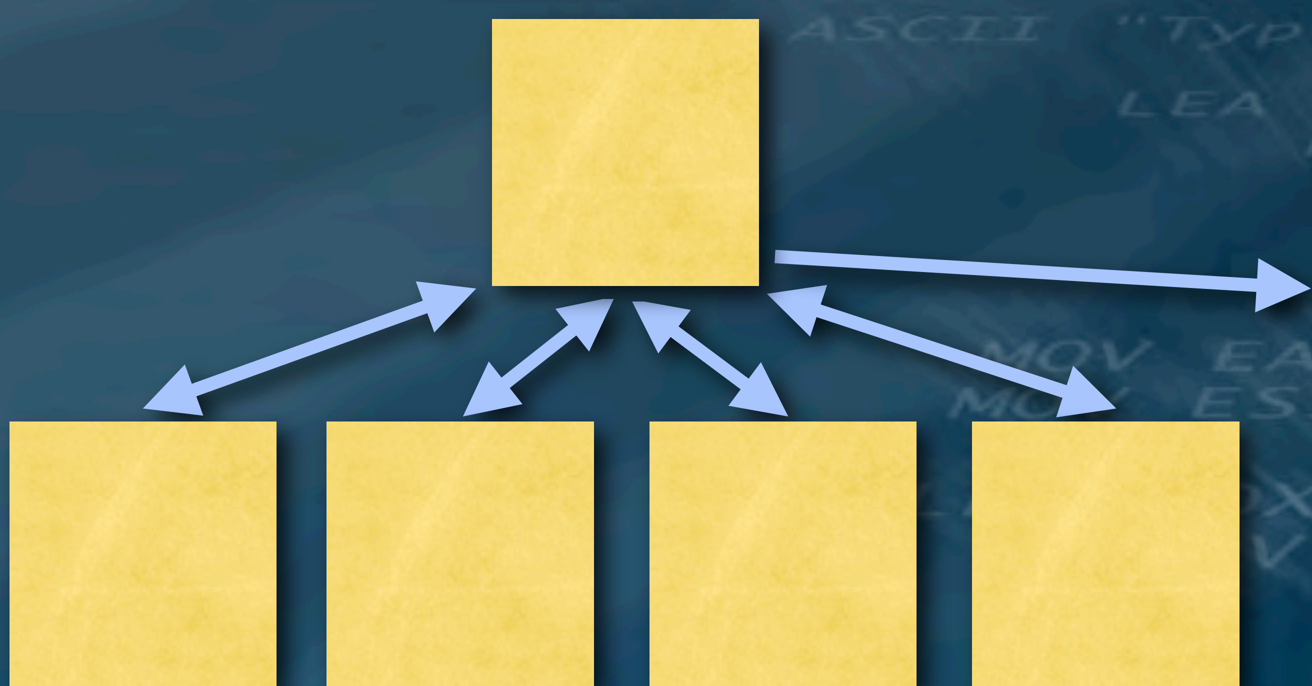
Control-Flow Obfuscation

- Control-flow flattening involves, quite literally, flattening the graphical representation of your program.
- Typically you have a top-down flow with program graphs. With flattening, you cause a central piece of code to control the flow of the program.
- Control-flow obfuscation is employed by bin/crackmes/leetkey.exe

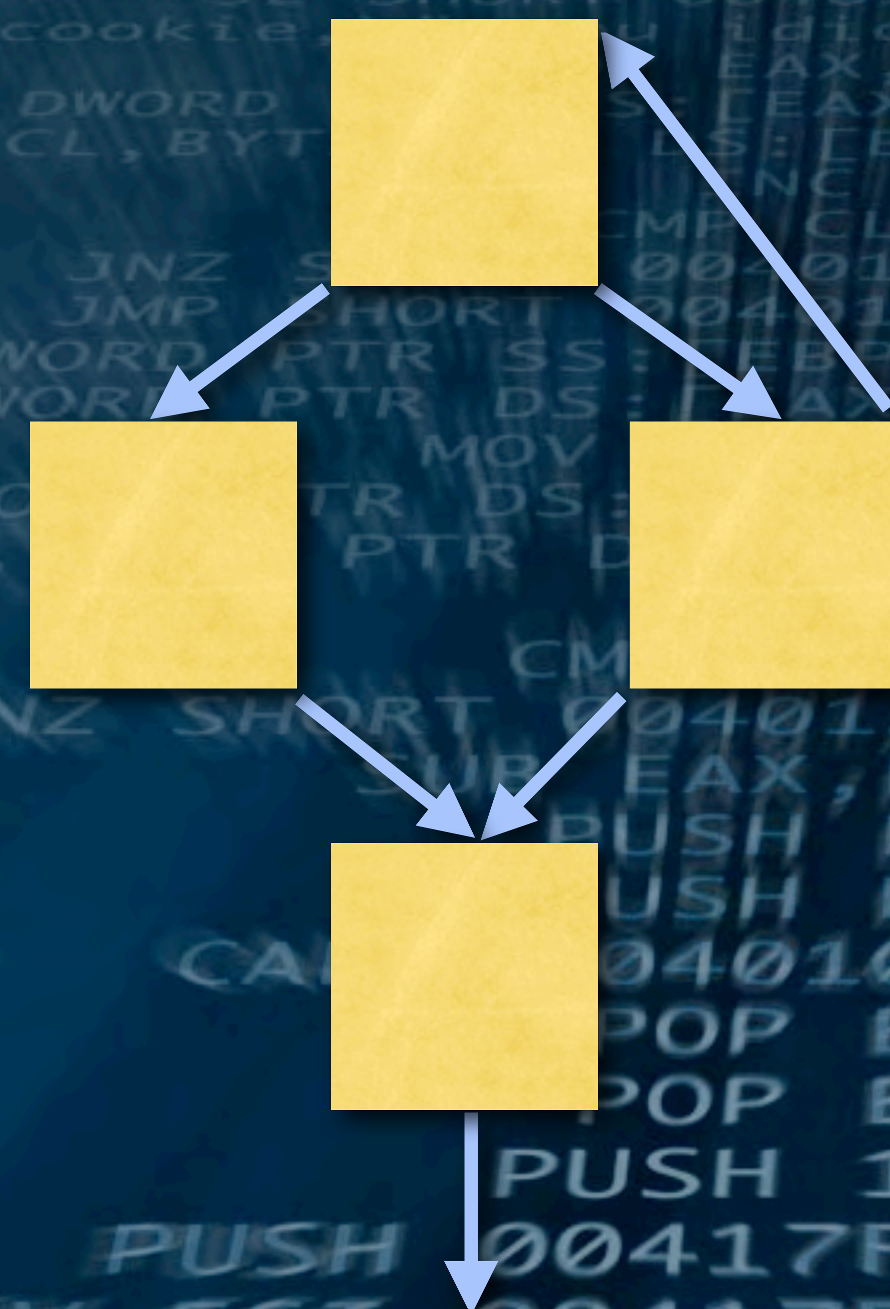
Binary Obfuscation

Control-Flow Obfuscation

Flattened:

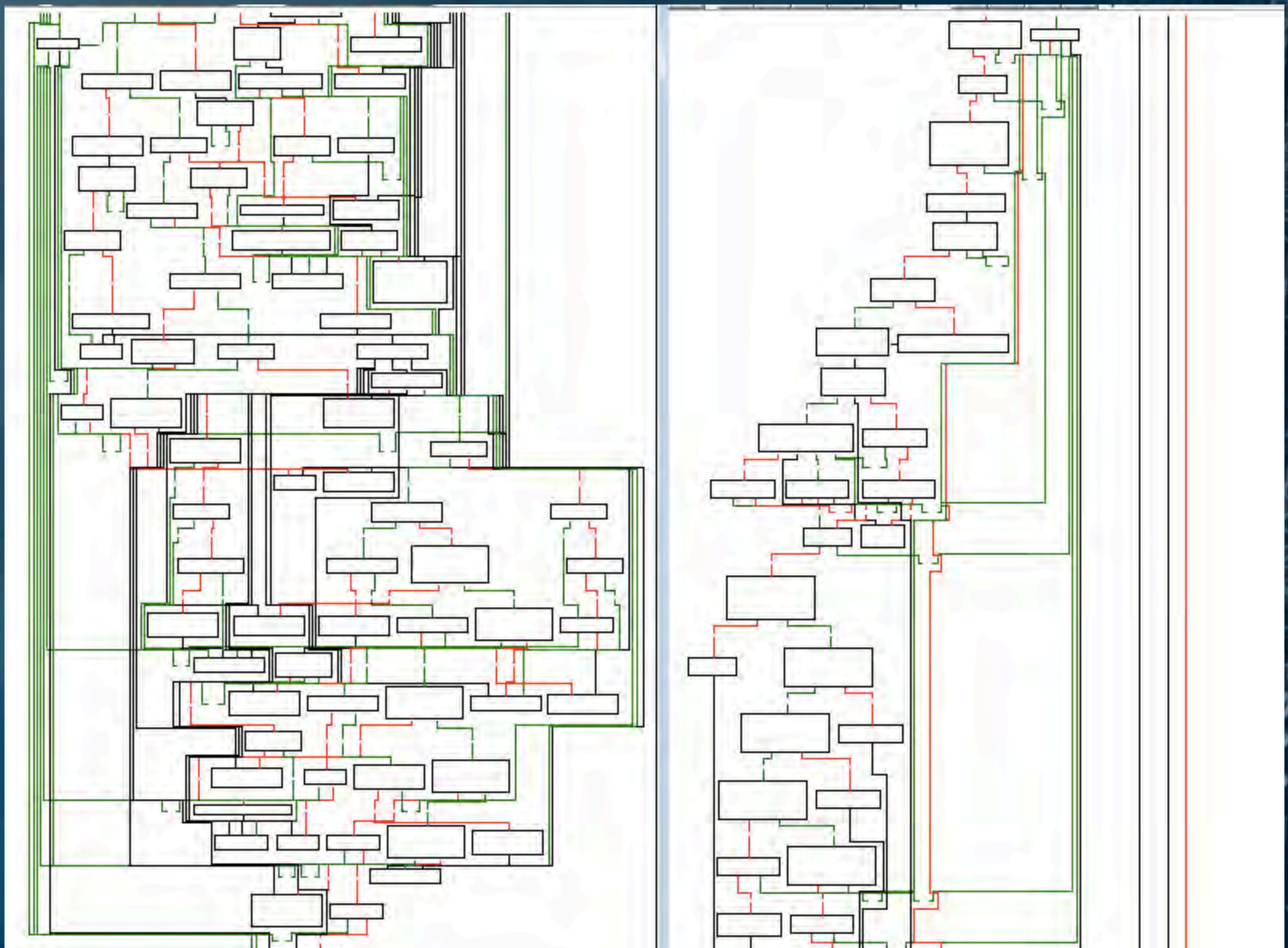


Normal:



Binary Obfuscation

Control-Flow Obfuscation



Binary Obfuscation

Control-Flow Obfuscation

```
doThis();
doThat();
doMore();
```

→

```
int x=2;
sw: switch(x) {
   case 0: doThat();
   x = 1;
   goto sw;
   case 1: doMore();
   break;
   case 2: doThis();
   x = 0;
   goto sw;
}
```

Binary Obfuscation

Control-Flow Obfuscation

- This technique of obfuscation can be applied very creatively.
- See src/cflow-flatlist.c and src/cflow-flattree.c

Binary Obfuscation

Control-Flow Obfuscation

- Most programs are reducible-- meaning they can easily be optimized.
- If a program is irreducible, then it cannot be optimized, thus translating spaghetti code into spaghetti assembly.
- A good example by Madou et. al. is making a loop irreducible.
- See src/cflow-irreducible.c

Binary Obfuscation

Control-Flow Obfuscation

- Raising bogus exceptions is a common way for malware to obfuscate and frustrate reverse engineering.
- This is easily accomplished by setting up a `try` block, intentionally triggering the exception, then resuming at the caught section.
- For Linux, you can do the same with signals.
- See src/cflow-exceptions.cpp

Binary Obfuscation

Control-Flow Obfuscation

```
try {
    volatile int trigger=20;
    doThis();
    doThat();
    /* trigger divide-by-zero exception */
    trigger=trigger/(trigger-trigger);
    neverExecutes();
} catch (...) {
    doMore();
    doTonsMore();
}
```

Data Obfuscation

Binary Obfuscation

Data Obfuscation

- Data obfuscation takes a little more care than control-flow obfuscation.
- The data must be obfuscated before the compilation process, then de-obfuscated at run-time.
- If the data is not obfuscated before run-time, dead-code analysis is made trivial and your obfuscation is useless.

Binary Obfuscation

Data Obfuscation

- One of the more obvious techniques is to encrypt your strings.
- Even though strings don't technically lead to knowledge of the program, it can help aide in reverse-engineering more often than you think.

Binary Obfuscation

Data Obfuscation

- Recall the explanation of `volatile`:

```
volatile int x;
x = 7;
x <<= 2;
x *= 2;
x -= 12;
x += (x*x)<<2;
printf("%d\n", x);
```

- With enough annoyances, this can be used to frustrate analysis.

Binary Obfuscation

Data Obfuscation

- Data aggregation can be used to make dead-code analysis confusing.

```
char aggr[7] = “fboaor”;
char foo[3], bar[3];
int i;
for (i=0;i<3;++i) {
   foo[i]=aggr[i*2];
   bar[i]=aggr[i*2+1];
}
/* foo = “foo” / bar = “bar” */
```

Binary Obfuscation

Data Obfuscation

- Functions in the PLT/IAT are certainly considered data.
- To prevent dead-code analysis from discovering our library calls, we can easily "create" functions at run-time by using system calls such as `LoadLibrary` and `GetProcAddress` (Windows) and `dlopen` and `dlsym` (Linux).
- See src/data-loadlib.c, src/data-dlopen.c and src/mdl.cpp

Poor Man's Packer

How to simulate a packer in a humorous manner.

Binary Obfuscation

Poor Man's Packer

- Combines control-flow and data obfuscation to cause all sorts of headaches.
- Revolves around compiling, copying data and applying function pointers to obfuscated or encrypted data.
- See bin/crackmes/manifest.exe
  - If you have problems with this binary, ask a DC949 member what the group motto is.

Binary Obfuscation

Poor Man's Packer

- Compile
- Disassemble
- Copy bytes of function, make an array
- Apply encryption, aggregation, etc.
- Recompile
- Decipher at run-time
- Cast as function-pointer
- Execute
- See src/pmp-concept.c

Binary Obfuscation

Poor Man's Packer

- Problems
  - Functions are broken because they are no longer in the PLT/IAT.
  - Data offsets are completely messed up.
  - Functions in C++ objects cause segmentation faults (due to broken thiscall).
  - Compiler might change calling conventions.
  - `void` pointers are scary.

Binary Obfuscation

Poor Man's Packer

- If you pass a data structure containing data required by the function (function offsets, strings, etc.), you can circumvent the issue caused by relative jumps and offsets.
- This also applies to method pointers and C++ objects.
- This gives you the opportunity to dynamically add and remove necessary program data as you see fit.

Binary Obfuscation

Poor Man's Packer

- Be sure your calling conventions match after each step of compilation and byte-copying!
- cdecl is the calling convention used by vararg functions such as `printf`.
- fastcall and stdcall should be fine for all other functions.
- Mismatched calling conventions will cause headaches and segmentation faults.

Binary Obfuscation

Poor Man's Packer

- Why is this beneficial?
  - Ultimate control of all data
  - Code is still portable and executable
  - Adds a bizarre layer of obfuscation
  - When done enough, severely obfuscates source

Binary Obfuscation

Poor Man's Packer

- Why does this suck?
  - Makes binaries huge if you don't compress your functions due to enlarged data-sections
  - Takes a lot of work to accomplish
  - It can be extremely frustrating to craft the write code with the right keywords with full optimization

Additional Info

Some stuff to help you out with obfuscation

Binary Obfuscation

Tools

- Code transformers
  - TXL (txl.ca)
  - SUIF (suif.standford.edu)

- TXL and SUIF are used to transform source-code by a certain set of given rules (such as regular expressions).

Binary Obfuscation

Sources

- M. Madou, B. Anckaert, B. De Bus, K. De Bosschere, J. Cappaert, and B. Preneel, "On the Effectiveness of Source Code Transformations for Binary Obfuscation"
- B. M. Prasad, T. Chiueh, "A Binary Rewriting Defense against Stack based Buffer Overflows"
- C. I. Popov, S. Debray, G. Andrews, "Binary Obfuscation Using Signals"

The End