Int. Secure Systems Lab
Technical University Vienna

Internet Security

Reverse Engineering and Binary Analysis

Adrian Dabrowski, Georg Merzdovnik, Aljosha Judmayer,

Johanna Ullrich, Christian Kudera

News from the Lab

- Challenges 4 Still running for one week
  - 31 Solves already
  - Fastet solve: “Slicon Dead” with 2:52:46

CTF Intro Meetup: Reversing

Int. Secure Systems Lab
Technical University Vienna

- Today we will have another Meetup
  - 17:30 @ EI3A
  - Intro to
    - Reverse Engineering,
    - disassembly
    - software side channel attacks

https://w0y.at/blog.html

News from the Field

- Efail (https://efail.de/)
  - Problem of Interaction between Mailclients and Encryption Software

    “Our advice, which mirrors that of the researchers, is to immediately disable and/or uninstall tools that automatically decrypt PGP-encrypted email. Until the flaws described in the paper are more widely understood and fixed, users should arrange for the use of alternative end-to-end secure channels, such as Signal, and temporarily stop sending and especially reading PGP-encrypted email.”

    https://www.eff.org/deeplinks/2018/05/attention-pgp-users-new-vulnerabilities-require-you-take-action-now
- Attack works by injecting HTML into email and thereby exfiltrating content

News from the Field

- Next Day:
- Code injection Attack in Signal Desktop
  - https://twitter.com/ortegaalfredo/status/995017143002509313
  - Based on Electron (Based on [outdate] Chromium)
  - Attack allows execution of javascript without interaction

Overview

- Introduction

- Reverse engineering
  - Intel x86 Assembler Primer
  - static vs. dynamic analysis techniques
  - anti-reverse engineering

- Malicious code analysis

Introduction

- Reverse engineering
  - process of analyzing a system
  - understand its structure and functionality
  - used in different domains (e.g., consumer electronics)

- Software reverse engineering
  - understand architecture (from source code)
  - extract source code (from binary representation)
  - change code functionality (of proprietary program)
  - understand message exchange (of proprietary protocol)

Reverse Engineering

- Application areas
  - copy (steal) technology
  - allow for interoperability
    - Samba (SMB protocol), WINE (Windows API), OpenOffice (MS Office), NTFS (file system structure), ...
  - circumvent copy protection or access restrictions
    - program cracking, creation of license key-generators (keygens)

- Techniques
  - static approaches
  - dynamic approaches

Reverse Engineering

- Static techniques
  - read documentation
  - read source code
  - analyze binary for strings, symbols, and library functions
  - disassemble binary image

- Dynamic techniques
  - observe interaction with environment
    - file system, network, registry
  - observe interaction with operating system
    - system calls
  - debug process

Reverse Engineering

Static Techniques

Static Techniques

- Gathering program information

```
$ cat test.c

#include <stdio.h>

int main (int argc, char **argv)
{
  if (argc == 2 && strcmp(argv[1], "correctSerial") == 0)
  {
    printf("do something useful\n");
  }
  else
  {
    printf("usage: %s <correct-serial>\n", argv[0]);
  }
  return 0;
}
```

Static Techniques

- Gathering program information
  - strings that the binary contains
    - `strings` command

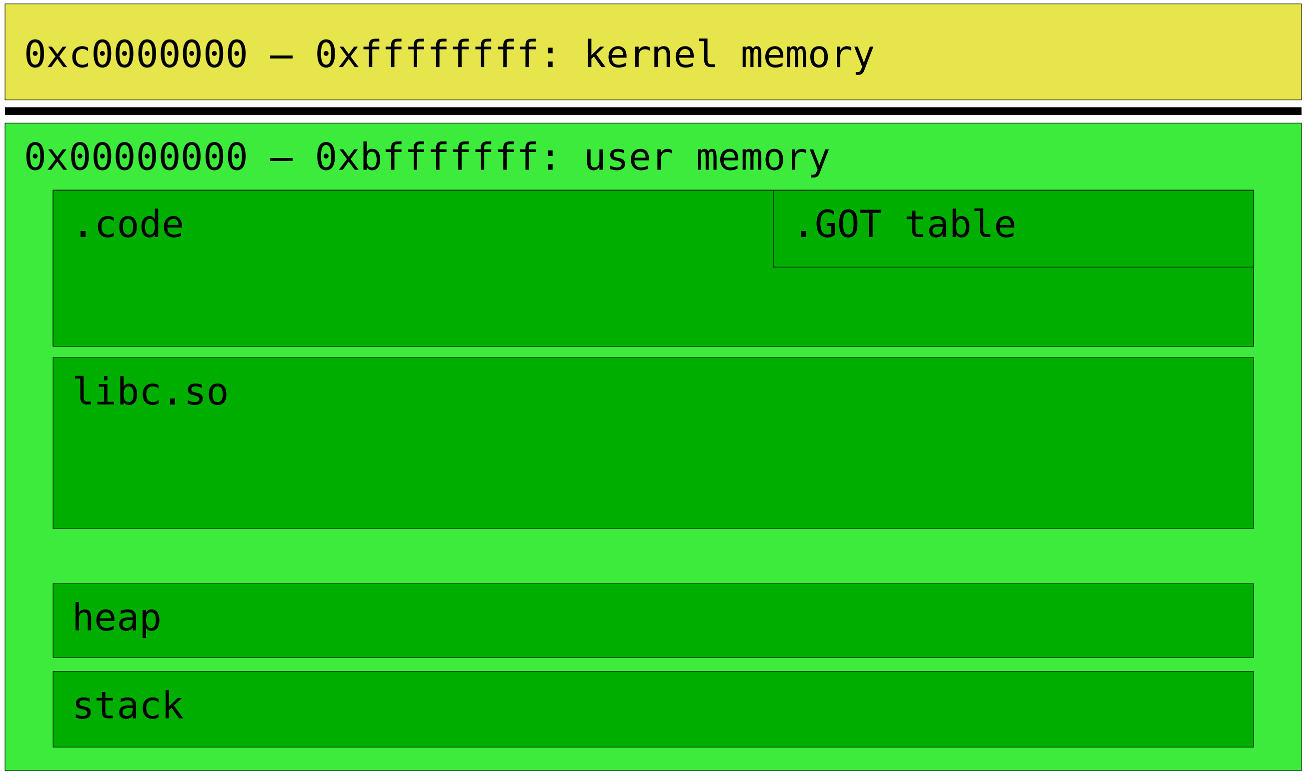
```
$ strings test

/lib64/ld-linux-x86-64.so.2   GLIBC_2.2.5
libm.so.6                     fff.
__gmon_start__                fffff.
_Jv_RegisterClasses           l$ L
libc.so.6                     t$(L
puts                          |$0H
printf                        correctSerial
strcmp                        do something useful
libc_start_main               usage: %s <correct-serial>
```

Static Techniques

- Gathering program information
  - library functions that were used
    - easy when program is dynamically linked

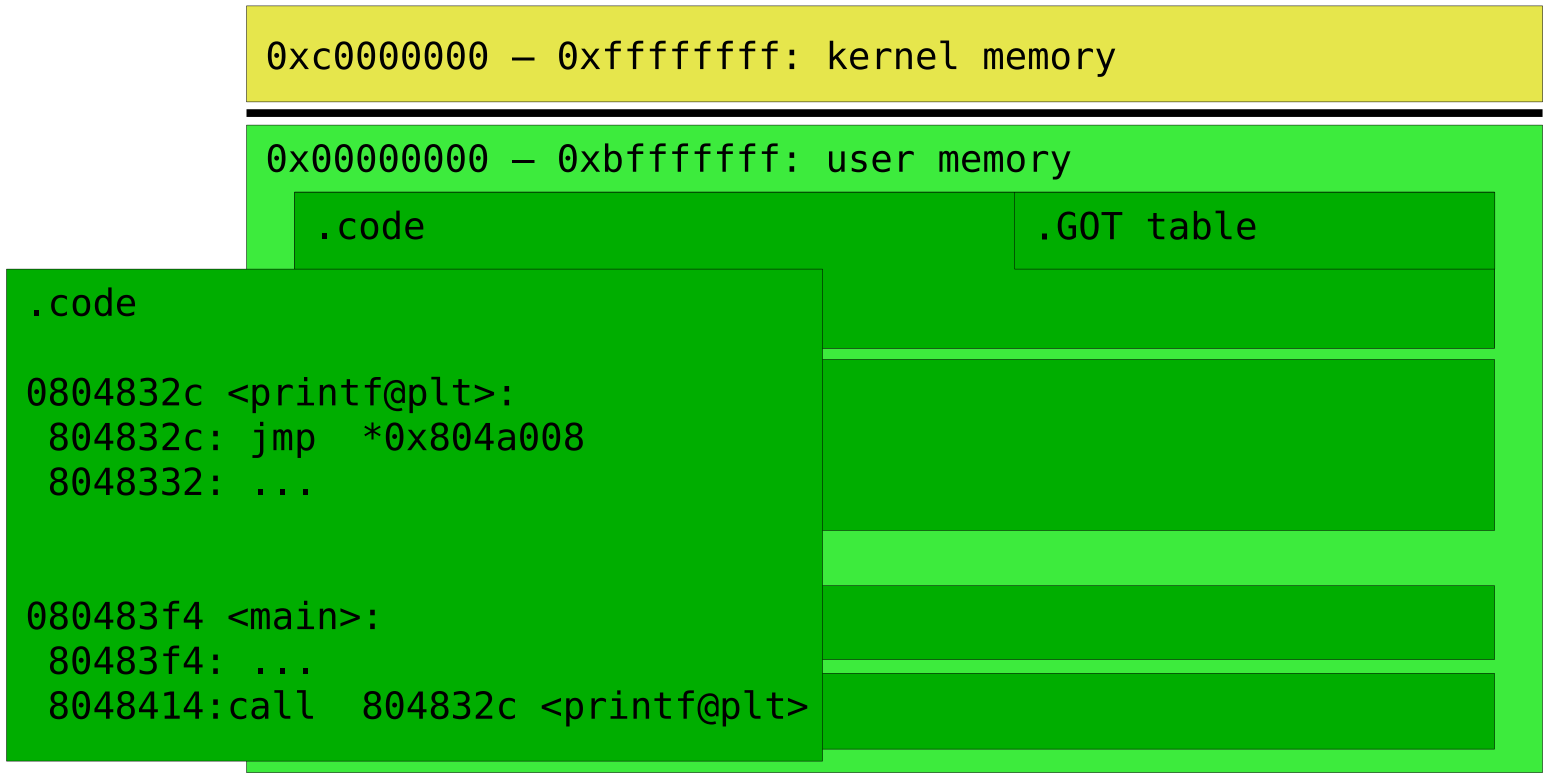
Shared Libraries

- Process layout (32 bit systems)


0xc0000000 – 0xffffffff: kernel memory
0x00000000 – 0xbfffffff: user memory
.code
.GOT table
libc.so
heap
stack

Shared Libraries

- Process layout (32 bit systems)

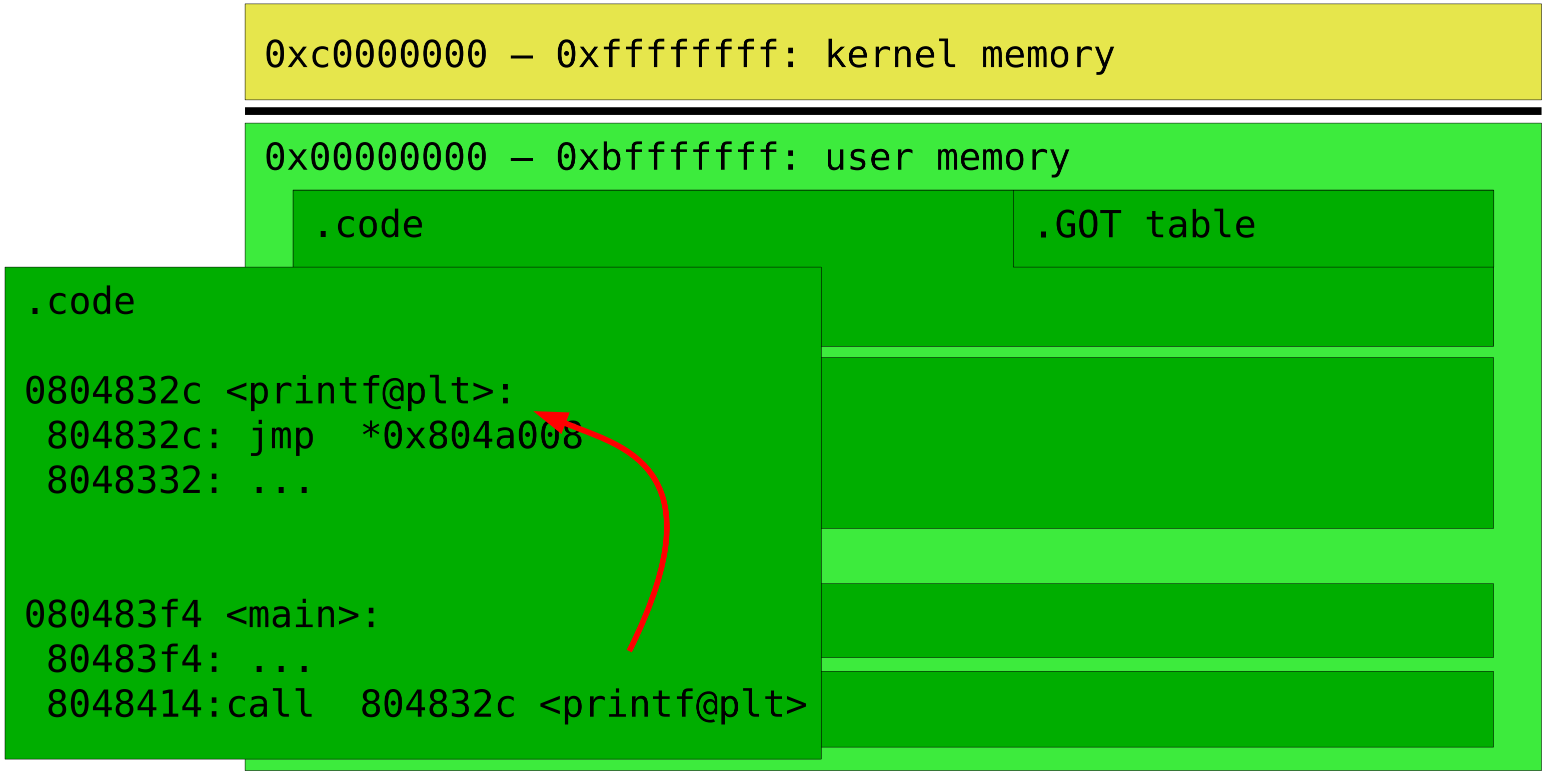

0xc0000000 – 0xffffffff: kernel memory
0x00000000 – 0xbfffffff: user memory
.code
.GOT table
.code

```
0804832c <printf@plt>:
 804832c: jmp  *0x804a008
 8048332: ...


080483f4 <main>:
 80483f4: ...
 8048414:call  804832c <printf@plt>
```

Shared Libraries

Int. Secure Systems Lab
Technical University Vienna

- Process layout (32 bit systems)



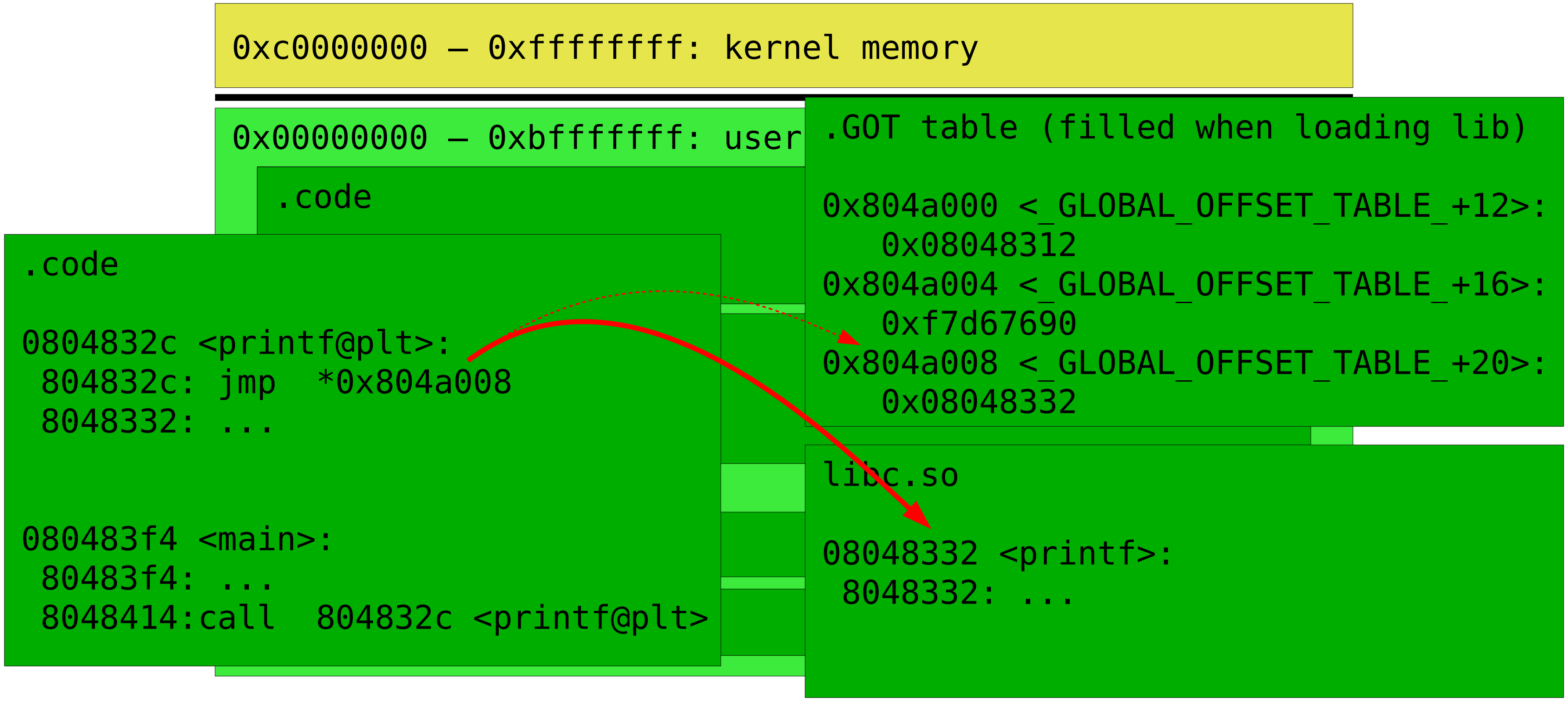
0xc0000000 – 0xffffffff: kernel memory
0x00000000 – 0xbfffffff: user memory
.code
.GOT table

```
.code

0804832c <printf@plt>:
 804832c: jmp  *0x804a008
 8048332: ...


080483f4 <main>:
 80483f4: ...
 8048414:call  804832c <printf@plt>
```

Shared Libraries

Int. Secure Systems Lab
Technical University Vienna

- Process layout (32 bit systems)



```
0xc0000000 – 0xffffffff: kernel memory
```

```
0x00000000 – 0xbfffffff: user
```

```
.code
```

```
.code

0804832c <printf@plt>:
 804832c: jmp   *0x804a008
 8048332: ...


080483f4 <main>:
 80483f4: ...
 8048414:call  804832c <printf@plt>
```

```
.GOT table (filled when loading lib)

0x804a000 <_GLOBAL_OFFSET_TABLE_+12>:
   0x08048312
0x804a004 <_GLOBAL_OFFSET_TABLE_+16>:
   0xf7d67690
0x804a008 <_GLOBAL_OFFSET_TABLE_+20>:
   0x08048332
```

```
libc.so

08048332 <printf>:
 8048332: ...
```

Static Techniques

- Gathering program information
  - library functions that were used
    - easy when program is dynamically linked
    - use `ldd` to find imported libraries

```
$ ldd test
        linux-vdso.so.1 =>  (0x00007fff701ff000)
        libm.so.6 => /lib/libm.so.6 (0x00007f3f2dd94000)
        libc.so.6 => /lib/libc.so.6 (0x00007f3f2da25000)
        /lib64/ld-linux-x86-64.so.2 (0x00007f3f2e018000)
```

Static Techniques

- Gathering program information
  - library functions that were used
    - easy when program is dynamically linked
    - use `objdump` to find linked functions

```
$ objdump -R test
...
DYNAMIC RELOCATION RECORDS
OFFSET           TYPE              VALUE
0000000000601000 R_X86_64_JUMP_SLOT  printf
0000000000601008 R_X86_64_JUMP_SLOT  puts
0000000000601018 R_X86_64_JUMP_SLOT  strcmp
```

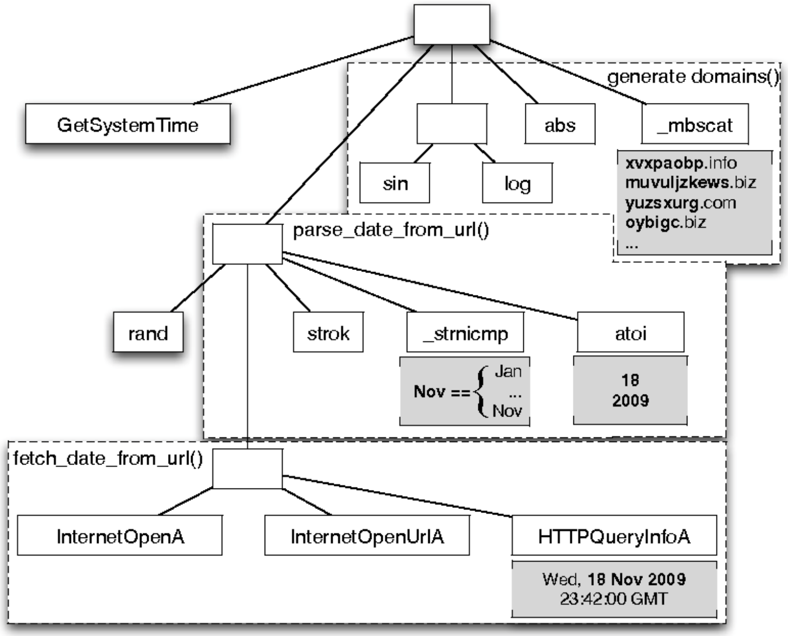
    - more difficult when program is statically linked
    - use function fingerprints
      - support through tools: `IDA` or `dress`

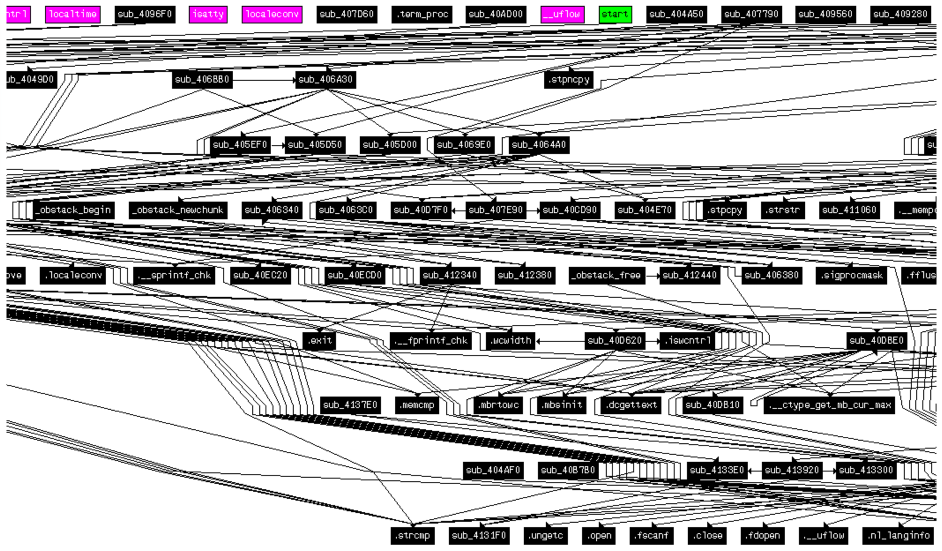
Static Techniques

- Gathering program information
  - program symbols
    - used for debugging (and linking)
    - function names (with start addresses)
    - global variables
    - can be removed with `strip`
    - use `nm` to display symbol information
  - function call trees
    - draw a graph that shows which function calls which other function
    - get an idea of program structure

Static Techniques

- Gathering program information
  - function call trees
    - Conficker.A domain name generation algorithm (DGA)





Static Techniques

- Disassembly
  - process of translating binary stream into machine instructions
- Different levels of difficulty
  - depending on ISA (instruction set architecture)
- Instructions can have
  - fixed length
    - more efficient to decode for processor
    - RISC processors (SPARC, MIPS)
  - variable length
    - use less space for common instructions
    - CISC processors (Intel x86)

Static Techniques

- Fixed length instructions
  - easy to disassemble
  - each address is a multiple of the instruction length
  - even if code contains data (or junk), all program instructions are found

- Variable length instructions
  - difficult to disassemble
  - start addresses of instructions not known in advance
  - disassembler can be desynchronized with respect to actual code
    - force disassembler to output incorrect instructions
    - obfuscation attack
  - different strategies
    - linear sweep disassembler (i.e. obdjump)
    - recursive traversal disassembler (i.e. IDA Pro)

Intel x86 Assembler Primer

- Assembler Language
  - human-readable form of machine instructions
  - must understand the hardware architecture, memory model, and stack

- What does this Instruction do?

```
MOV Reg1, Reg2
```

Intel x86 Assembler Primer

- Assembler Language
  - human-readable form of machine instructions
  - must understand the hardware architecture, memory model, and stack

- What does this Instruction do?

```
MOV Reg1, Reg2
```

- It depends: **AT&T** syntax vs. **Intel** syntax

AT&T vs. Intel Syntax

AT&T	Intel
`mnemonic source(s), destination`	`mnemonic destination, source(s)`
`MOV src, dest`	**`MOV dest, src`**
Constants: prefixed with $	No prefix
Hexadecimal numbers: start with `0x`	hexadecimal numbers: start with `0x`
Registers: prefixed with %	Registers: No prefix
Memory access is of form `displacement(%base, %index, scale)` where the result address is `displacement + %base + %index*scale`	Memory access is of form `<size> [disp + index*4 + base]` where the result address is `disp + index*4 + base` Example: `dword [ebx + ecx*4 + mem_location]`

AT&T vs. Intel: Example

```
$ objdump -M att -d /bin/ls

...
push %ebp
xor %ecx,%ecx
mov %esp,%ebp
sub $0x8,%esp
mov %ebx,(%esp)
mov 0x8(%ebp),%ebx
mov %esi,0x4(%esp)
mov 0xc(%ebp),%esi
mov (%ebx),%edx
mov 0x4(%ebx),%eax
xor 0x4(%esi),%eax
xor (%esi),%edx
or %edx,%eax
je 8049c60 <exit@plt+0x13c>
...
```

```
$ objdump -M intel -d /bin/ls

...
push ebp
xor ecx,ecx
mov ebp,esp
sub esp,0x8
mov DWORD PTR [esp],ebx
mov ebx,DWORD PTR [ebp+0x8]
mov DWORD PTR [esp+0x4],esi
mov esi,DWORD PTR [ebp+0xc]
mov edx,DWORD PTR [ebx]
mov eax,DWORD PTR [ebx+0x4]
xor eax,DWORD PTR [esi+0x4]
xor edx,DWORD PTR [esi]
or eax,edx
je 8049c60 <exit@plt+0x13c>
...
```

Intel x86 Assembler Primer

- Identifying Syntax
  - Intel: **MOV dest, src**
  - AT&T: **MOV src, dest**
  - Find out yourself:
    - Look out for read-only elements, constants → match them as source

- IDA Pro, Windows usually use Intel Syntax
- objdump, Unix Systems prefer AT&T syntax
  - Usually you will find a switch/argument to change the syntax)

Registers

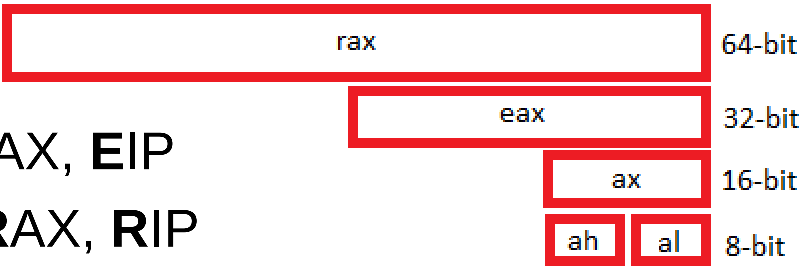
- Local variables of processor
  - Efficient access
    - No delays compared to loading from RAM/Memory
  - Are accessed by name in assembly instructions
  - Different categories
    - General-purpose register (GPR)
    - Special-purpose regsiters (SPR)
    - Vector registers
    - Data registers

- Instruction Pointer
  - The EIP register contains the address of the next instruction to be executed if no branching is done

General-purpose registers

- Eight 32-bit general purpose registers (GPR)
  - can be used for calculations, temporary storage of values, …
  - `%eax, %ebx, %ecx, %edx, %esi, %edi, %esp, %ebp`
    - `%esp` - stack pointer
    - `%ebp` - frame/base pointer

- Registers Extensions
  - "**E**" prefix for 32bit variants → **E**AX, **E**IP
  - "**R**" prefix for 64 bit variants → **R**AX, **R**IP
    - Additional GPRs for 64 bit: **R8** → **R15**

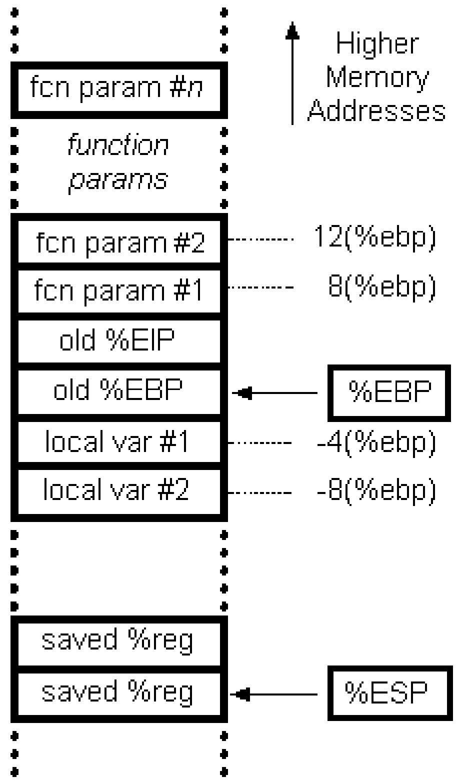


Status register (EFLAGS register)

- The EFLAGS is a 32-bit register used as a collection of bits representing Boolean values to store the results of operations and the state of the processor
  - **CF: Carry Flag** Set if the last arithmetic operation carried (addition) or borrowed (subtraction) a bit beyond the size of the register
  - **PF: Parity Flag** Set if the number of set bits in the least significant byte is a multiple of 2
  - **ZF: Zero Flag** Set if the result of an operation is Zero
  - **SF: Sign Flag** Set if the result of an operation is negative
  - … and many more

Intel x86 Assembler Primer

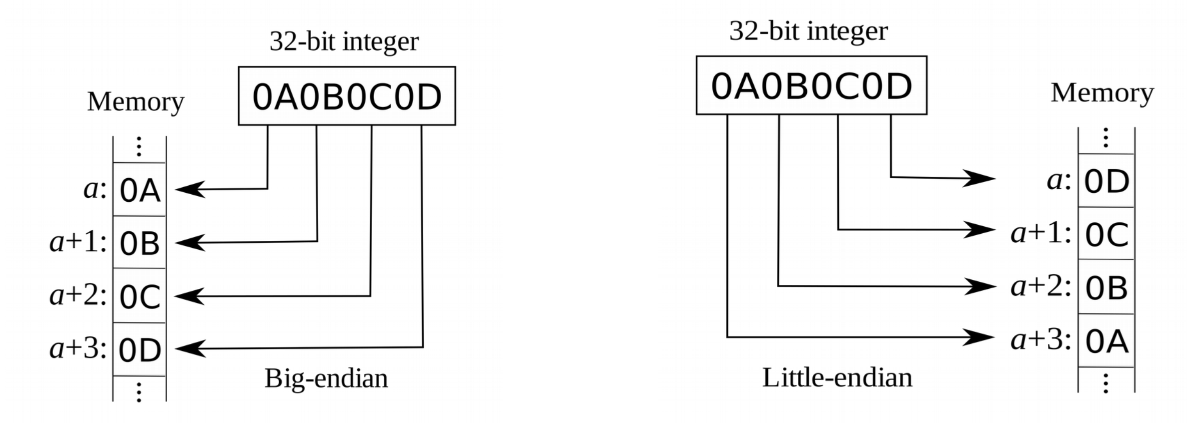
- Stack
  - managed by stack pointer (%esp) and frame pointer (%ebp)
  - used for
    - function arguments
    - function return address
    - local arguments



Intel x86 Assembler Primer

- Endianness/ Byte ordering
  - important for multi-byte values (e.g., four byte long value)
  - Intel Architecture uses *little endian* ordering
  - how to represent `0x11223344` in memory (at `addr`)?

```
0x010004 (addr)      :   0x44
0x010005 (addr+1)    :   0x33
0x010006 (addr+2)    :   0x22
0x010007 (addr+3)    :   0x11
```



Intel x86 Assembler Primer

- Important mnemonics (instructions)

`mov`	data transfer
`push/pop`	top of stack manipulation
`add/sub`	arithmetic
`cmp/test`	compare two values and set control flags
`je/jne`	conditional jump depending on control flags (branch)
`jmp`	unconditional jump

- Numerical representation
  - Binary (0,1): 10011100
    - Prefix: **0b**10011100 ← Unix (both Intel and AT&T)
    - Suffix: 10011100**b** ← Traditional Intel syntax
  - Hexadecimal (0...F): “**0x**” vs “**h**”
    - Prefix: **0x**ABCD1234 ← Easy to notice
    - Suffix: ABCD1234**h** ← Number or literal? (Usually Syntax highlighting will help out)

Intel x86 Assembler Primer

- Addressing modes
  - Direct: **MOV EAX, [10h]**
    - Copy value located at address 10h
  - Indirect: **MOV EAX, [EBX]**
    - Copy value pointed to by register BX
  - Indexed: **MOV AL, [EBX + ECX * 4 + 10h]**
    - Copy value from array (BX[4 * CX + 0x10])
  - Pointers can be associated to type
    - **MOV AL, byte ptr [BX]**

- For 64bit you can also read/use **RIP** for addressing
  - Useful for Position-independent code (and shellcode)

Intel x86 Assembler Primer

- `if` statement

```
#include <stdio.h>

int main(int argc, char **argv)
{
  int a;

  if(a < 0) {
    printf("A < 0\n");
  }
  else {
    printf("A >= 0\n");
  }
}
```

```
.LC0:
        .string "A < 0\n"
.LC1:
        .string "A >= 0\n"
.globl main
        .type   main, @function
main:
        [ function prologue ]
        cmp    $0, -4(%ebp) /* s = a - 0*/
        jns     .L2          /* if sign bit is not
                                         set */
        mov    $.LC0, (%esp)
        call    printf
        jmp     .L3
.L2:
        mov    $.LC1, (%esp)
        call    printf
.L3:
        leave
        ret
```

Intel x86 Assembler Primer

- while statement

```
#include <stdio.h>

int main(int argc, char **argv)
{
    int i;

    i = 0;
    while(i < 10)
    {
        printf("%d\n", i);
        i++;
    }
}
```

```
.LC0:
        .string "%d\n"
main:
        [ function prologue ]
        mov    $0, -4(%ebp)
.L2:
        cmp    $9, -4(%ebp)
        jle     .L4    /* Jump if less or equal */
        jmp     .L3
.L4:
        mov    -4(%ebp), %eax
        mov    %eax, 4(%esp)
        mov    $.LC0, (%esp)
        call    printf
        lea    -4(%ebp), %eax   /* Load Address */
        inc    (%eax)
        jmp     .L2
.L3:
        leave
        ret
```

Intel x86 Assembler Primer

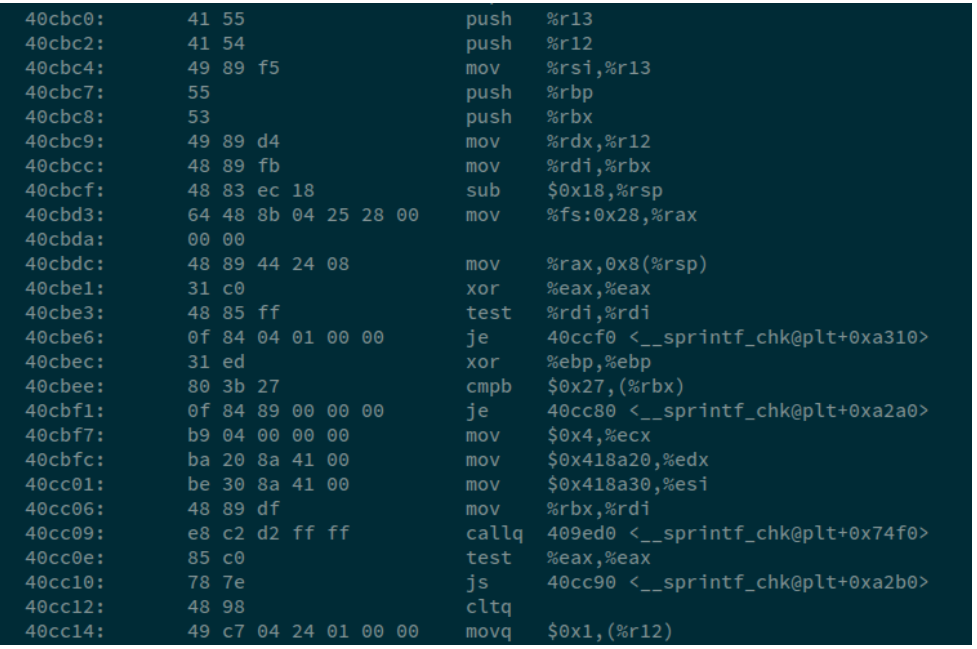
- Calling Conventions
  - Standard for passing arguments to function calls
  - Caller and Callee need to agree
  - Enforced by compiler
  - Important for 3rd party library usage
  - Different styles ↔ different Pros/cons

Intel x86 Assembler Primer

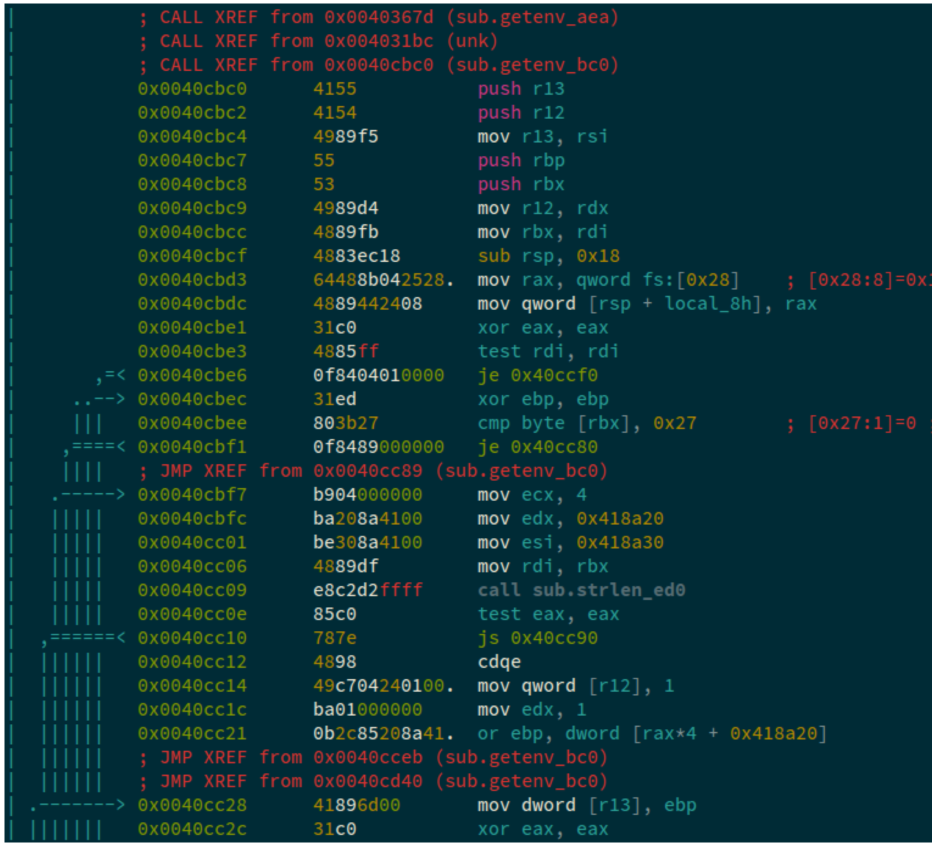
- **System V AMD64 ABI**
  - Used on *NIX systems
  - Arguments (Integer/Pointer) passed in
    - RDI, RSI, RDX, RCX, R8, R9
  - System calls use R10 instead of RCX
  - Floating Point arguments passed in XMM registers
  - All Additional Arguments are passed on stack
  - Microsoft x64 calling convention similar
    - Uses: RCX, RDX, R8, R9

Disassembly

Int. Secure Systems Lab
Technical University Vienna



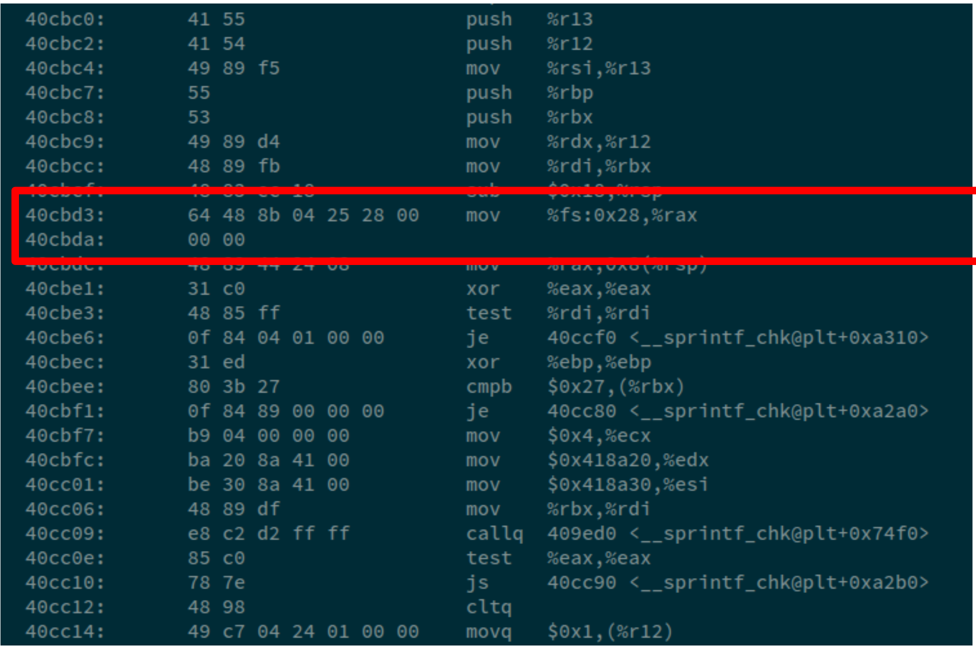
```
40cbc0:       41 55                   push   %r13
40cbc2:       41 54                   push   %r12
40cbc4:       49 89 f5                mov    %rsi,%r13
40cbc7:       55                      push   %rbp
40cbc8:       53                      push   %rbx
40cbc9:       49 89 d4                mov    %rdx,%r12
40cbcc:       48 89 fb                mov    %rdi,%rbx
40cbcf:       48 83 ec 18             sub    $0x18,%rsp
40cbd3:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
40cbda:       00 00
40cbdc:       48 89 44 24 08          mov    %rax,0x8(%rsp)
40cbe1:       31 c0                   xor    %eax,%eax
40cbe3:       48 85 ff                test   %rdi,%rdi
40cbe6:       0f 84 04 01 00 00       je     40ccf0 <__sprintf_chk@plt+0xa310>
40cbec:       31 ed                   xor    %ebp,%ebp
40cbee:       80 3b 27                cmpb   $0x27,(%rbx)
40cbf1:       0f 84 89 00 00 00       je     40cc80 <__sprintf_chk@plt+0xa2a0>
40cbf7:       b9 04 00 00 00          mov    $0x4,%ecx
40cbfc:       ba 20 8a 41 00          mov    $0x418a20,%edx
40cc01:       be 30 8a 41 00          mov    $0x418a30,%esi
40cc06:       48 89 df                mov    %rbx,%rdi
40cc09:       e8 c2 d2 ff ff          callq  409ed0 <__sprintf_chk@plt+0x74f0>
40cc0e:       85 c0                   test   %eax,%eax
40cc10:       78 7e                   js     40cc90 <__sprintf_chk@plt+0xa2b0>
40cc12:       48 98                   cltq
40cc14:       49 c7 04 24 01 00 00    movq   $0x1,(%r12)
```



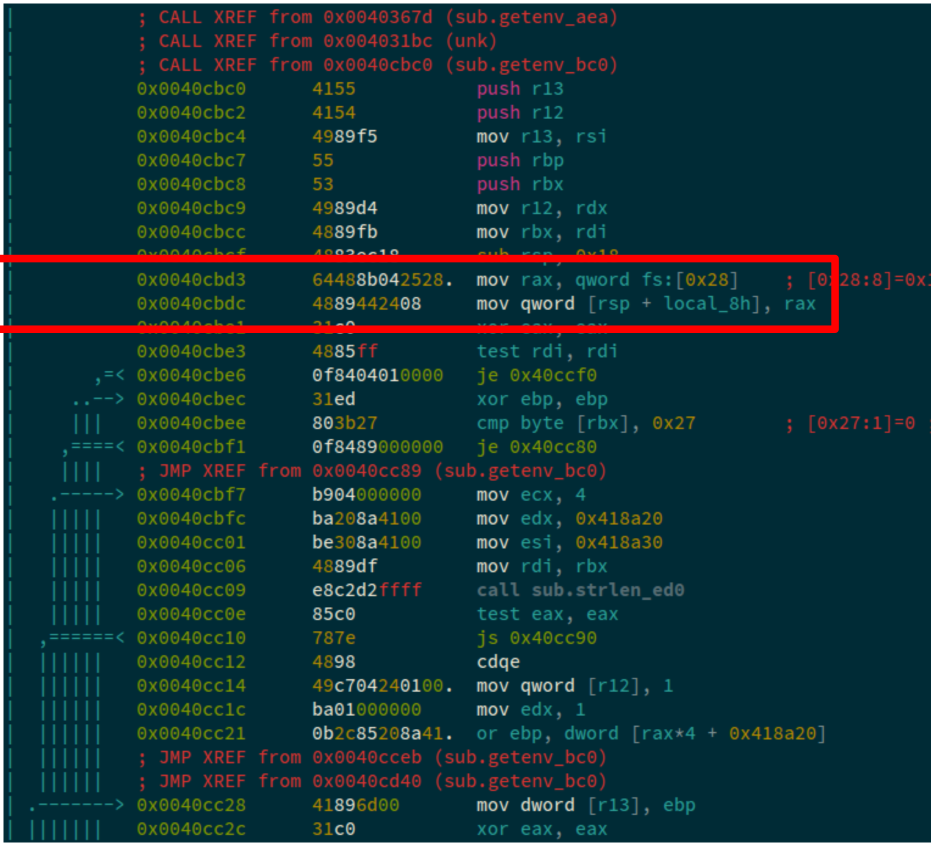
```
|              ; CALL XREF from 0x0040367d (sub.getenv_aea)
|              ; CALL XREF from 0x004031bc (unk)
|              ; CALL XREF from 0x0040cbc0 (sub.getenv_bc0)
|              0x0040cbc0      4155           push r13
|              0x0040cbc2      4154           push r12
|              0x0040cbc4      4989f5         mov r13, rsi
|              0x0040cbc7      55             push rbp
|              0x0040cbc8      53             push rbx
|              0x0040cbc9      4989d4         mov r12, rdx
|              0x0040cbcc      4889fb         mov rbx, rdi
|              0x0040cbcf      4883ec18       sub rsp, 0x18
|              0x0040cbd3      64488b042528.  mov rax, qword fs:[0x28]     ; [0x28:8]=0x
|              0x0040cbdc      4889442408     mov qword [rsp + local_8h], rax
|              0x0040cbe1      31c0           xor eax, eax
|              0x0040cbe3      4885ff         test rdi, rdi
|         ,=< 0x0040cbe6      0f8404010000   je 0x40ccf0
|       ..--> 0x0040cbec      31ed           xor ebp, ebp
|       |||   0x0040cbee      803b27         cmp byte [rbx], 0x27         ; [0x27:1]=0
|      ,====< 0x0040cbf1      0f8489000000   je 0x40cc80
|      ||||   ; JMP XREF from 0x0040cc89 (sub.getenv_bc0)
|     .-----> 0x0040cbf7      b904000000     mov ecx, 4
|     |||||   0x0040cbfc      ba208a4100     mov edx, 0x418a20
|     |||||   0x0040cc01      be308a4100     mov esi, 0x418a30
|     |||||   0x0040cc06      4889df         mov rdi, rbx
|     |||||   0x0040cc09      e8c2d2ffff     call sub.strlen_ed0
|     |||||   0x0040cc0e      85c0           test eax, eax
|    ,======< 0x0040cc10      787e           js 0x40cc90
|    ||||||   0x0040cc12      4898           cdqe
|    ||||||   0x0040cc14      49c704240100.  mov qword [r12], 1
|    ||||||   0x0040cc1c      ba01000000     mov edx, 1
|    ||||||   0x0040cc21      0b2c85208a41.  or ebp, dword [rax*4 + 0x418a20]
|    ||||||   ; JMP XREF from 0x0040cceb (sub.getenv_bc0)
|    ||||||   ; JMP XREF from 0x0040cd40 (sub.getenv_bc0)
|   .-------> 0x0040cc28      41896d00       mov dword [r13], ebp
|   |||||||   0x0040cc2c      31c0           xor eax, eax
```

Disassembly

Int. Secure Systems Lab
Technical University Vienna



```
40cbc0:       41 55                   push   %r13
40cbc2:       41 54                   push   %r12
40cbc4:       49 89 f5                mov    %rsi,%r13
40cbc7:       55                      push   %rbp
40cbc8:       53                      push   %rbx
40cbc9:       49 89 d4                mov    %rdx,%r12
40cbcc:       48 89 fb                mov    %rdi,%rbx
40cbcf:       48 83 ec 18             sub    $0x18,%rsp
40cbd3:       64 48 8b 04 25 28 00    mov    %fs:0x28,%rax
40cbda:       00 00
40cbdc:       48 89 44 24 08          mov    %rax,0x8(%rsp)
40cbe1:       31 c0                   xor    %eax,%eax
40cbe3:       48 85 ff                test   %rdi,%rdi
40cbe6:       0f 84 04 01 00 00       je     40ccf0 <__sprintf_chk@plt+0xa310>
40cbec:       31 ed                   xor    %ebp,%ebp
40cbee:       80 3b 27                cmpb   $0x27,(%rbx)
40cbf1:       0f 84 89 00 00 00       je     40cc80 <__sprintf_chk@plt+0xa2a0>
40cbf7:       b9 04 00 00 00          mov    $0x4,%ecx
40cbfc:       ba 20 8a 41 00          mov    $0x418a20,%edx
40cc01:       be 30 8a 41 00          mov    $0x418a30,%esi
40cc06:       48 89 df                mov    %rbx,%rdi
40cc09:       e8 c2 d2 ff ff          callq  409ed0 <__sprintf_chk@plt+0x74f0>
40cc0e:       85 c0                   test   %eax,%eax
40cc10:       78 7e                   js     40cc90 <__sprintf_chk@plt+0xa2b0>
40cc12:       48 98                   cltq
40cc14:       49 c7 04 24 01 00 00    movq   $0x1,(%r12)
```



```
|           ; CALL XREF from 0x0040367d (sub.getenv_aea)
|           ; CALL XREF from 0x004031bc (unk)
|           ; CALL XREF from 0x0040cbc0 (sub.getenv_bc0)
|           0x0040cbc0      4155           push r13
|           0x0040cbc2      4154           push r12
|           0x0040cbc4      4989f5         mov r13, rsi
|           0x0040cbc7      55             push rbp
|           0x0040cbc8      53             push rbx
|           0x0040cbc9      4989d4         mov r12, rdx
|           0x0040cbcc      4889fb         mov rbx, rdi
|           0x0040cbcf      4883ec18       sub rsp, 0x18
|           0x0040cbd3      64488b042528.  mov rax, qword fs:[0x28]    ; [0x28:8]=0x
|           0x0040cbdc      4889442408     mov qword [rsp + local_8h], rax
|           0x0040cbe1      31c0           xor eax, eax
|           0x0040cbe3      4885ff         test rdi, rdi
|       ,=< 0x0040cbe6      0f8404010000   je 0x40ccf0
|     ..--> 0x0040cbec      31ed           xor ebp, ebp
|     |||   0x0040cbee      803b27         cmp byte [rbx], 0x27        ; [0x27:1]=0
|    ,====< 0x0040cbf1      0f8489000000   je 0x40cc80
|    ||||   ; JMP XREF from 0x0040cc89 (sub.getenv_bc0)
|   .-----> 0x0040cbf7      b904000000     mov ecx, 4
|   |||||   0x0040cbfc      ba208a4100     mov edx, 0x418a20
|   |||||   0x0040cc01      be308a4100     mov esi, 0x418a30
|   |||||   0x0040cc06      4889df         mov rdi, rbx
|   |||||   0x0040cc09      e8c2d2ffff     call sub.strlen_ed0
|   |||||   0x0040cc0e      85c0           test eax, eax
|  ,======< 0x0040cc10      787e           js 0x40cc90
|  ||||||   0x0040cc12      4898           cdqe
|  ||||||   0x0040cc14      49c704240100.  mov qword [r12], 1
|  ||||||   0x0040cc1c      ba01000000     mov edx, 1
|  ||||||   0x0040cc21      0b2c85208a41.  or ebp, dword [rax*4 + 0x418a20]
|  ||||||   ; JMP XREF from 0x0040cceb (sub.getenv_bc0)
|  ||||||   ; JMP XREF from 0x0040cd40 (sub.getenv_bc0)
| .-------> 0x0040cc28      41896d00       mov dword [r13], ebp
| |||||||   0x0040cc2c      31c0           xor eax, eax
```

Disassembly

- Linear sweep disassembler
  - start at beginning of code (.text) section
  - disassemble one instruction after the other
  - assume that well-behaved compiler tightly packs instructions
  - `objdump -d` uses this approach

- Obfuscation Attack
  - insert data (or junk) between instructions and let control flow jump over this garbage
  - disassembler gets confused

```
jmp L1
.short 0x4711
L1:
xor %eax, %eax
...

ret
```

```
4004cf:   eb 02                          jmp    4004d3
4004d1:   11 47                          <junk>

4004d3:   31 c0                          xor    %eax,%eax
4004d5:   b8 00 00 00 00                 mov    $0x0,%eax
4004da:   c9                             leave
4004db:   c3                             ret
```

CORRECT

Disassembly

- Linear sweep disassembler
  - start at beginning of code (.text) section
  - disassemble one instruction after the other
  - assume that well-behaved compiler tightly packs instructions
  - `objdump -d` uses this approach

- Obfuscation Attack
  - insert data (or junk) between instructions and let control flow jump over this garbage
  - disassembler gets confused

```
jmp L1
.short 0x4711
L1:
xor %eax, %eax
...

ret
```

```
4004cf:  eb 02                   jmp    4004d3
4004d1:  11 47 31                adc    %eax,0x31(%edi)

4004d4:  c0 b8 00 00 00 00 c9    sarb   $0xc9,0x0(%eax)


4004db:  c3                      ret
```

Disassembly

- Recursive traversal disassembler
  - aware of control flow
  - start at program entry point (e.g., determined by ELF header)
  - disassemble one instruction after the other, until branch or jump is found
  - recursively follow both (or single) branch (or jump) targets
  - not all code regions can be reached
    - indirect calls and indirect jumps
    - use a register to calculate target during run-time
  - for these regions, linear sweep is used
  - `IDA Pro` uses this approach

Disassembly

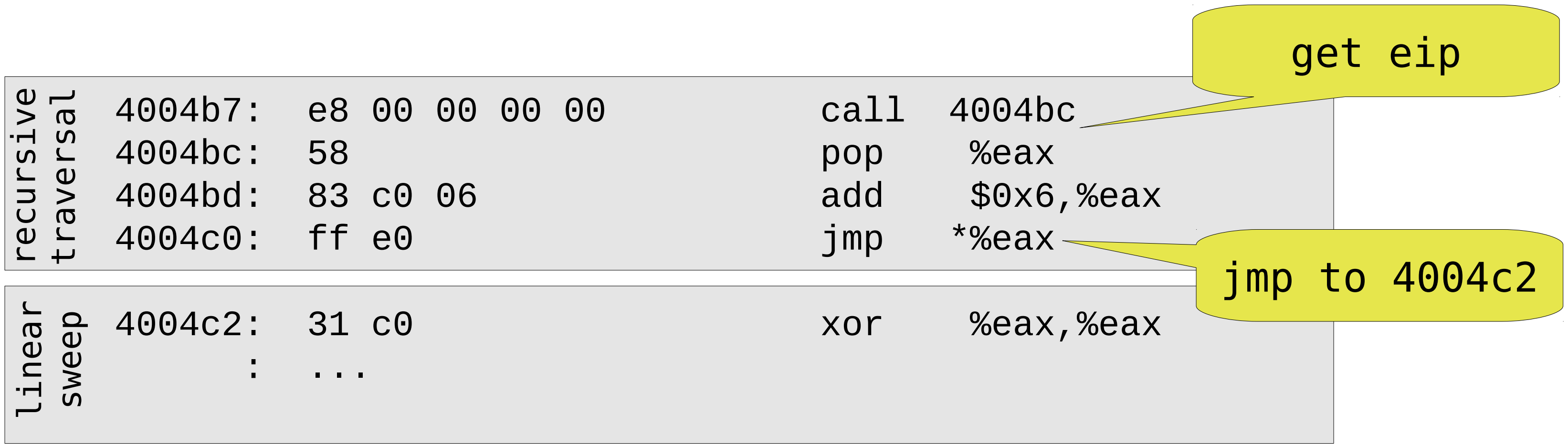
- Recursive traversal disassembler
- Obfuscation Attack
  - plain previous attack fails
  - replace direct jumps (calls) by indirect ones
  - force disassembler to revert to linear sweep, and then use previous attack

```
4004b7:   e8 00 00 00 00          call   4004bc
4004bc:   58                      pop     %eax
4004bd:   83 c0 06                add     $0x6,%eax
4004c0:   ff e0                   jmp    *%eax

4004c2:   31 c0                   xor     %eax,%eax
      :   ...
```

Disassembly

- Recursive traversal disassembler

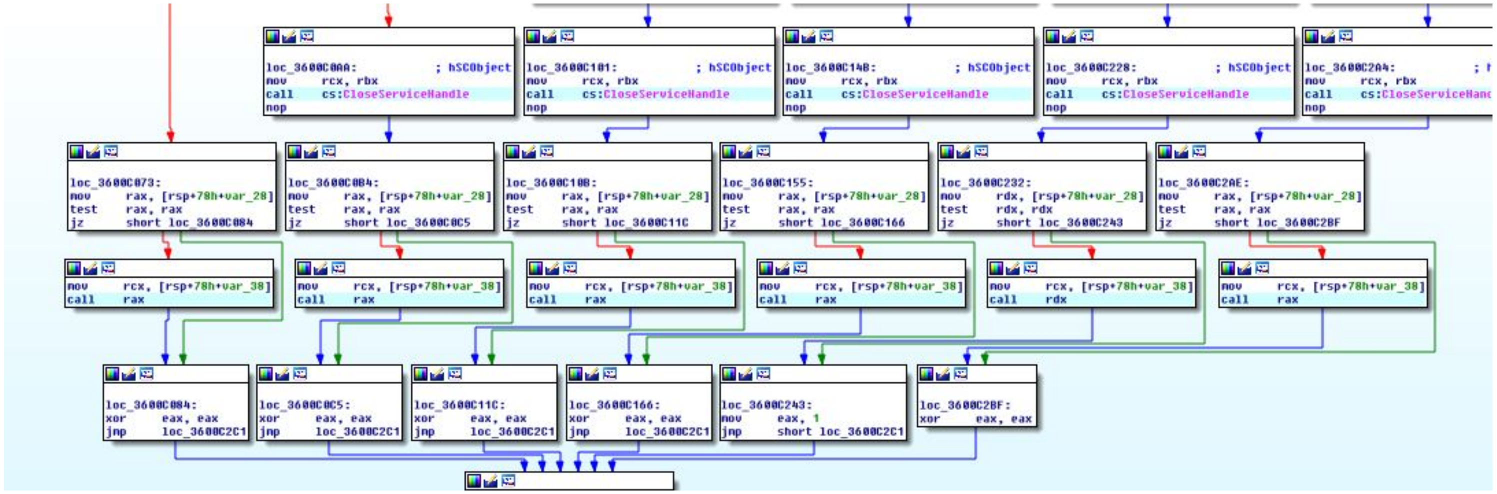
- Obfuscation Attack
  - plain previous attack fails
  - replace direct jumps (calls) by indirect ones
  - force disassembler to revert to linear sweep, and then use previous attack



```
4004b7:  e8 00 00 00 00          call  4004bc
4004bc:  58                      pop    %eax
4004bd:  83 c0 06                add    $0x6,%eax
4004c0:  ff e0                   jmp   *%eax

4004c2:  31 c0                   xor    %eax,%eax
      :  ...
```

Control Flow Graph

- Nodes are called basic blocks
- Edges represent possible flow of control from end of block to beginning of another block
- Control always enters at the beginning of a block and exits at the end



```
loc_3600C0AA:              ; hSCObject
mov     rcx, rbx
call    cs:CloseServiceHandle
nop
```

```
loc_3600C101:              ; hSCObject
mov     rcx, rbx
call    cs:CloseServiceHandle
nop
```

```
loc_3600C14B:              ; hSCObject
mov     rcx, rbx
call    cs:CloseServiceHandle
nop
```

```
loc_3600C228:              ; hSCObject
mov     rcx, rbx
call    cs:CloseServiceHandle
nop
```

```
loc_3600C2A4:              ; h
mov     rcx, rbx
call    cs:CloseServiceHand
nop
```

```
loc_3600C073:
mov     rax, [rsp+78h+var_28]
test    rax, rax
jz      short loc_3600C084
```

```
loc_3600C0B4:
mov     rax, [rsp+78h+var_28]
test    rax, rax
jz      short loc_3600C0C5
```

```
loc_3600C10B:
mov     rax, [rsp+78h+var_28]
test    rax, rax
jz      short loc_3600C11C
```

```
loc_3600C155:
mov     rax, [rsp+78h+var_28]
test    rax, rax
jz      short loc_3600C166
```

```
loc_3600C232:
mov     rdx, [rsp+78h+var_28]
test    rdx, rdx
jz      short loc_3600C243
```

```
loc_3600C2AE:
mov     rax, [rsp+78h+var_28]
test    rax, rax
jz      short loc_3600C2BF
```

```
mov     rcx, [rsp+78h+var_38]
call    rax
```

```
mov     rcx, [rsp+78h+var_38]
call    rax
```

```
mov     rcx, [rsp+78h+var_38]
call    rax
```

```
mov     rcx, [rsp+78h+var_38]
call    rax
```

```
mov     rcx, [rsp+78h+var_38]
call    rdx
```

```
mov     rcx, [rsp+78h+var_38]
call    rax
```

```
loc_3600C084:
xor     eax, eax
jmp     loc_3600C2C1
```

```
loc_3600C0C5:
xor     eax, eax
jmp     loc_3600C2C1
```

```
loc_3600C11C:
xor     eax, eax
jmp     loc_3600C2C1
```

```
loc_3600C166:
xor     eax, eax
jmp     loc_3600C2C1
```

```
loc_3600C243:
mov     eax, 1
jmp     short loc_3600C2C1
```

```
loc_3600C2BF:
xor     eax, eax
```

Bytecode Decompilation

- Bytecode Decompilation
  - Recreate program for interpreted languages

- Usually includes more information
  - Instructions are easier to reverse
  - Additional information in archives

- Examples for decompilers (just a small sample selection to get you started)
  - Python .pyc → uncompyle2
  - Java → Procyon/Luyten
  - .NET → ILSpy

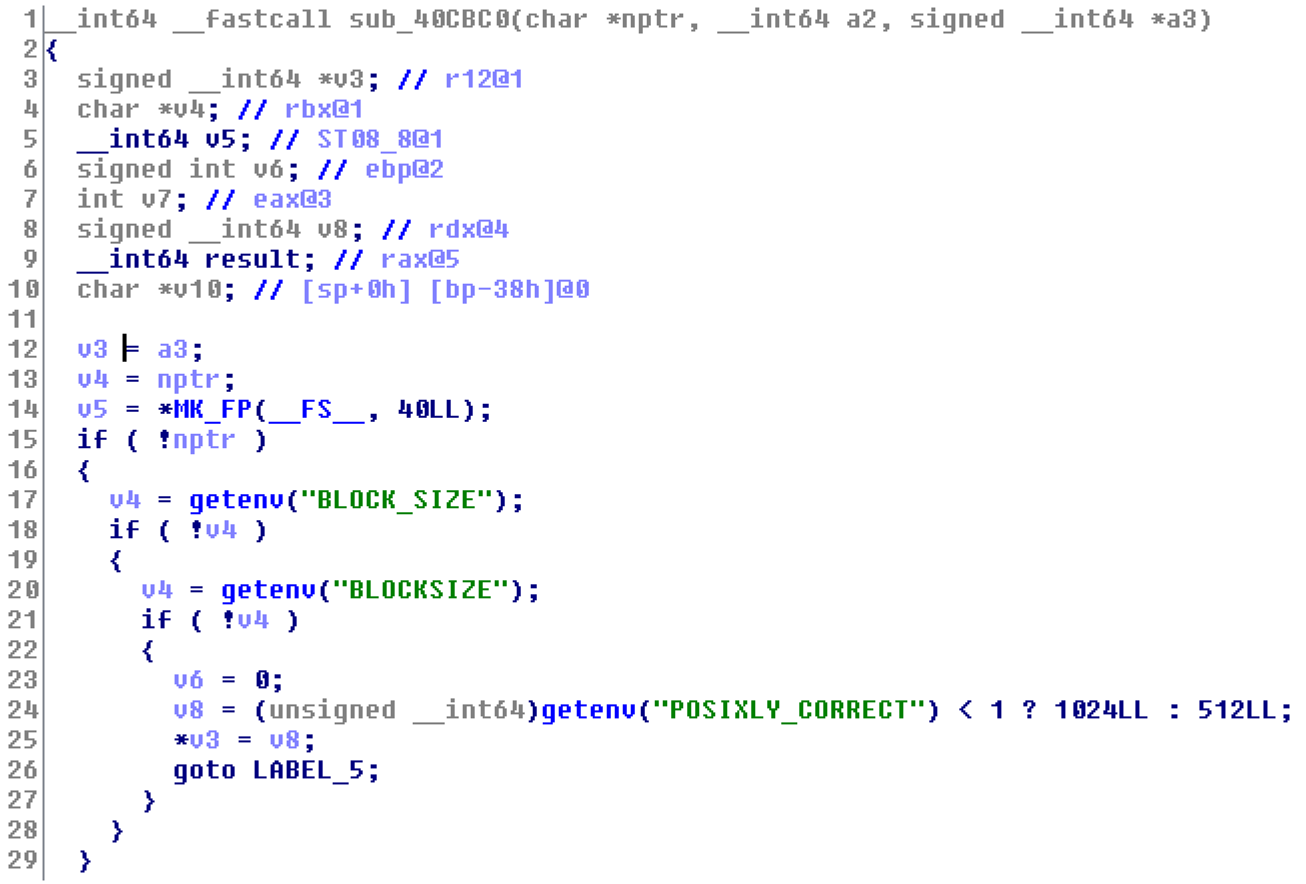
Binary Decompilation

- Binary Decompilation
  - Recreate high level representation of binary code
  - Usually C or C-like

- Faces several Problems
  - Optimizing compilers destroy structure
    - e.g. in-lining, loop unrolling,...
  - Type information is lost
  - Reconstruction of control flow...

- Still verry usefull, even if it provides incomplete results

Binary Decompilation



```
__int64 __fastcall sub_40CBC0(char *nptr, __int64 a2, signed __int64 *a3)
{
  signed __int64 *v3; // r12@1
  char *v4; // rbx@1
  __int64 v5; // ST08_8@1
  signed int v6; // ebp@2
  int v7; // eax@3
  signed __int64 v8; // rdx@4
  __int64 result; // rax@5
  char *v10; // [sp+0h] [bp-38h]@0

  v3 = a3;
  v4 = nptr;
  v5 = *MK_FP(__FS__, 40LL);
  if ( !nptr )
  {
    v4 = getenv("BLOCK_SIZE");
    if ( !v4 )
    {
      v4 = getenv("BLOCKSIZE");
      if ( !v4 )
      {
        v6 = 0;
        v8 = (unsigned __int64)getenv("POSIXLY_CORRECT") < 1 ? 1024LL : 512LL;
        *v3 = v8;
        goto LABEL_5;
      }
    }
  }
```

Reverse Engineering

Dynamic Techniques

Dynamic Techniques

- General information about process
  - `/proc file system`
  - `/proc/<pid>/` for a process with pid `<pid>`
  - interesting entries
    - `cmdline` (show command line)
    - `environ` (show environment)
    - `maps` (show memory map, *remember this for the challenges!!*)
    - `fd` (file descriptors held by program)
    - `exe` (program image)

- Interaction with the environment
  - file system
  - network

Dynamic Techniques

- File system interaction
  - `lsof`
  - lists all open files associated with processes
- Registry (Windows)
  - `regmon` (Sysinternals)
- Network interaction
  - check for open ports
    - processes that listen for requests or that have active connections
    - `ss (netstat [deprecated])`
    - also shows UNIX domain sockets used for IPC
  - check for actual network traffic
    - `tcpdump`
    - `wireshark`

Dynamic Techniques

- System calls
  - are at the boundary gates between user space and kernel
  - reveal much about a process' operation
  - `strace`
  - powerful tool that can also
    - follow child processes
    - decode more complex system call arguments
    - show signals
  - works via the `ptrace` interface

- Library functions
  - similar to system calls, but dynamically linked libraries
  - `ltrace`

Dynamic Techniques

- strace

```
$ strace echo "hi"

execve("/bin/echo", ["echo", "hi"], [/* 41 vars */])    = 0
brk(0)                                                  = 0xddb000
mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVA...)    = 0x7f54eac10000
...
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT (No such file or...)
open("/lib/libc.so.6", O_RDONLY)                        = 3
read(3, "\177ELF\2\1\1\0\0\0\0\0\0\0\0\3\0>\0\1\"..., 832) = 832
fstat(3, {st_mode=S_IFREG|0755, st_size=1490312, ...})  = 0
mmap(NULL, 3598344, PROT_READ|PROT_EXEC, ...)           = 0x7f54ea684000
mprotect(0x7f54ea7ea000, 2093056, PROT_NONE)            = 0
...
write(1, "hi\n", 3hi)                                   = 3
close(1)                                                = 0
munmap(0x7f54eaac1000, 4096)                            = 0
close(2)                                                = 0
exit_group(0)                                           = ?
```

Dynamic Techniques

- ltrace

```
$ ltrace echo "hi"

__libc_start_main(0x4013e0, 2, 0x7fffb3cfbe78, ...)
getenv("POSIXLY_CORRECT")                           = NULL
strrchr("echo", '/')                                = NULL
setlocale(6, "")                                    = "en_US.UTF-8"
bindtextdomain("coreutils", "/usr/share/locale")    = "/usr/share/locale"
textdomain("coreutils")                             = "coreutils"
...
fputs_unlocked(0x7fffb3cfc61e, 0x7f19cdc6a780, 0, 1, 0)      = 1
...
fclose(0x7f19cdc6a860)                              = 0
...
+++ exited (status 0) +++
```

Dynamic Techniques

- Execute program in a controlled environment
  - sandbox (virtual machine or emulator)
  - debugger
- Advantages
  - can inspect actual program behavior and data values
  - target of indirect jumps (or calls) can be observed
- Disadvantages
  - may accidentally launch attacks
  - anti-debugging mechanisms
  - not all possible traces (paths) can be seen

Dynamic Techniques

- Debugger
  - breakpoints to pause execution
    - when execution reaches a certain point (address)
    - when specified memory is access or modified
  - examine memory and CPU registers
  - modify memory and execution path

- Advanced features
  - attach comments to code
  - data structure and template naming
  - track high level logic
    - file descriptor tracking
  - function fingerprinting

Dynamic Techniques

- Debugger on x86 / Linux
  - use the `ptrace` interface

- `ptrace`
  - allows a process (parent) to monitor another process (child)
  - whenever the child process receives a signal, the parent is notified
  - parent can then
    - access and modify memory image (peek and poke commands)
    - access and modify registers
    - deliver signals
  - ptrace can also be used for system call monitoring

Dynamic Techniques

- Breakpoints
  - hardware breakpoints
  - software breakpoints

- Hardware breakpoints
  - special debug registers (e.g., Intel x86)
  - debug registers compared with PC at every instruction

- Software breakpoints
  - debugger inserts (overwrites) target address with an `int 0x03` instruction
  - interrupt causes signal SIGTRAP to be sent to process
  - debugger
    - gets control and restores original instruction
    - single steps to next instruction
    - re-inserts breakpoint

Dynamic Techniques

- Anti-debugging techniques
  - detect tracing
    - a process can be traced only once

```
if (ptrace(PTRACE_TRACEME, 0, 1, 0) < 0)
    exit(1);
```

  - detect breakpoints
    - look for `int 0x03` instructions

```
if ((*(unsigned *)((unsigned)<addr>+3) & 0xff)==0xcc)
    exit(1);
```

Dynamic Techniques

- Anti-debugging techniques (cont.)
  - checksum the code

```
if (checksum(text_segment) != valid_checksum)
    exit(1);
```

  - register signal handler for debug interrupt
    - force interrupt: parent will receive the signal

```
int dbg=1;
void my_handler(int signal) { dbg=0; };
int main(...) {
  signal(SIG_TRAP, my_handler);
  asm("int 0x03");
  if (dbg)
      exit(1);
```

Dynamic Techniques

- Reverse Debugging
  - Sometimes also called “Historical debugging” or “IntelliTrace” (Microsoft)

- Step through your program backwards in “time”
  - Usefull to identify the source of arguments/errors
  - You can use watchpoint/breakpoints as usual
- Gdb supports this since 7.0
  - Has to be activated explicitly in gdb
  - Imposes high runtime and memory overhead
    - Everything needs to be recorded
      - Registers, Old memory values,...

Malicious Code Analysis

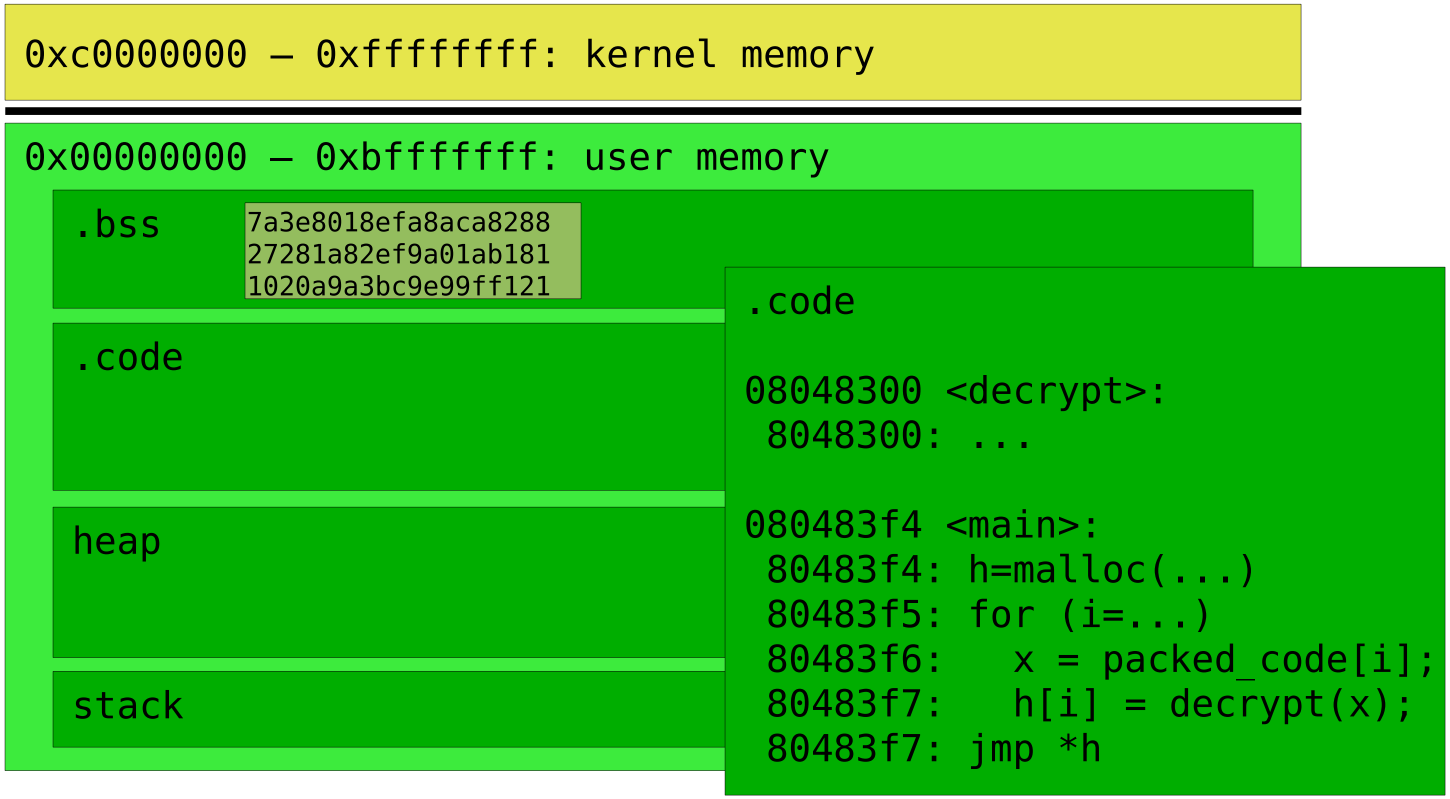
Malicious Code Analysis

Static analysis vs. dynamic analysis

- Static analysis
  - code is not executed
  - all possible branches can be examined (in theory)
  - quite fast
- Problems of static analysis
  - binary code typically contains very little information
    - functions, variables, type information, …
  - disassembly difficult (particularly for Intel x86 architecture)
  - obfuscated code
  - packed code, self-modifying code

Malicious Code Analysis

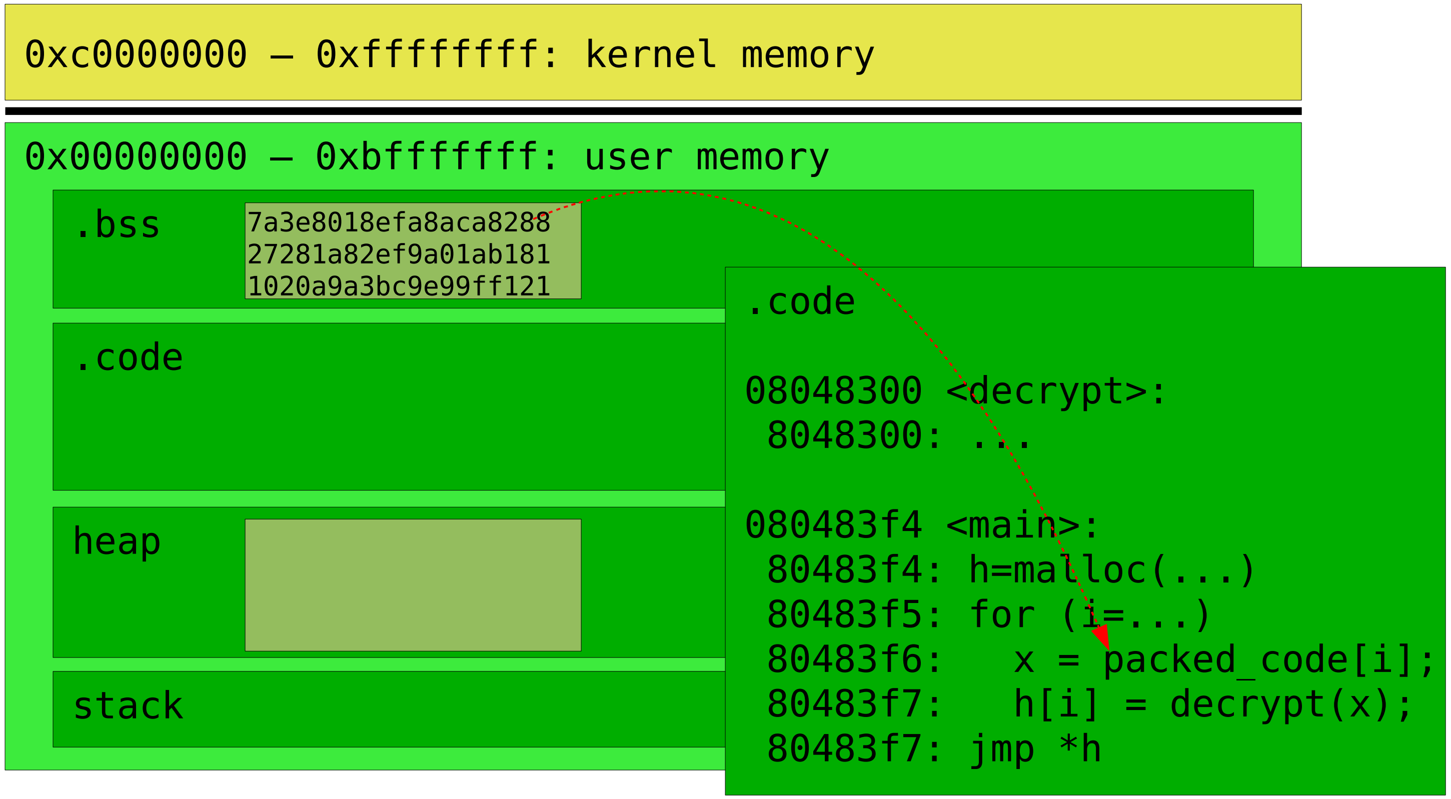
- Packed code (*dynamic unpacking*)



```
08048300 <decrypt>:
 8048300: ...

080483f4 <main>:
 80483f4: h=malloc(...)
 80483f5: for (i=...)
 80483f6:   x = packed_code[i];
 80483f7:   h[i] = decrypt(x);
 80483f7: jmp *h
```

Malicious Code Analysis

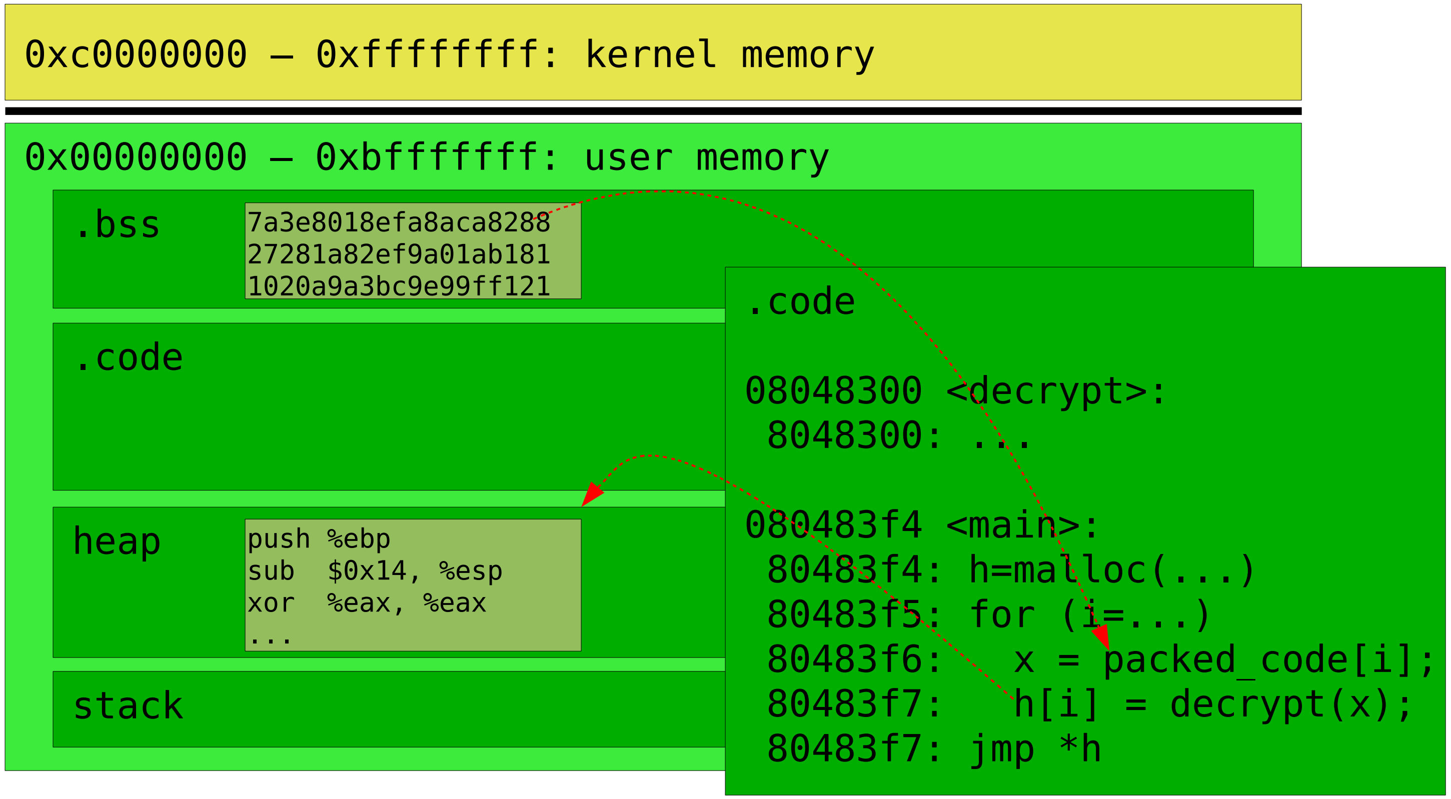
- Packed code (*dynamic unpacking*)



```
.code

08048300 <decrypt>:
 8048300: ...

080483f4 <main>:
 80483f4: h=malloc(...)
 80483f5: for (i=...)
 80483f6:   x = packed_code[i];
 80483f7:   h[i] = decrypt(x);
 80483f7: jmp *h
```

Malicious Code Analysis

- Packed code (*dynamic unpacking*)


0xc0000000 – 0xffffffff: kernel memory
0x00000000 – 0xbfffffff: user memory
.bss
.code
heap
stack

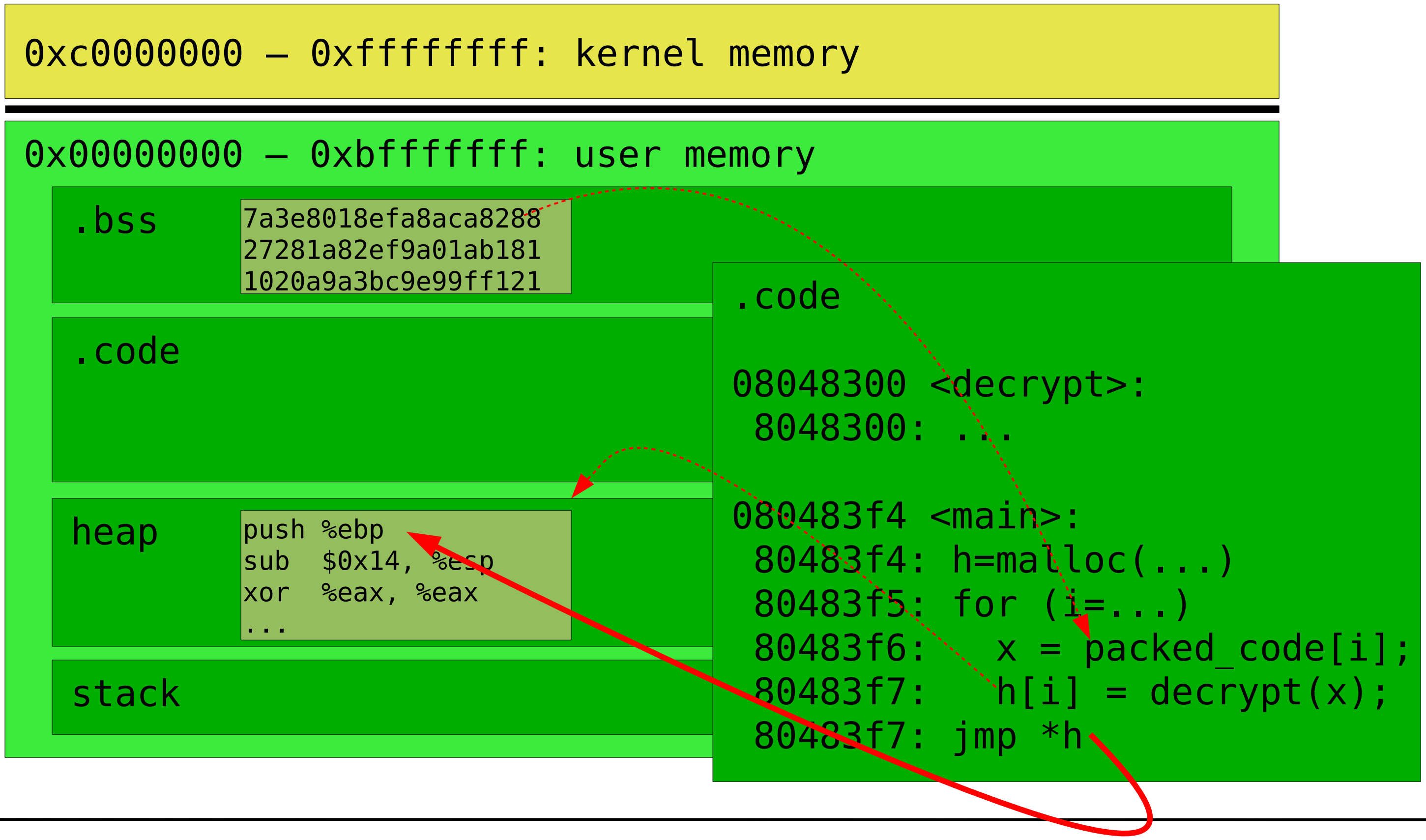
```
7a3e8018efa8aca8288
27281a82ef9a01ab181
1020a9a3bc9e99ff121
```

```
push %ebp
sub  $0x14, %esp
xor  %eax, %eax
...
```

```
.code

08048300 <decrypt>:
 8048300: ...

080483f4 <main>:
 80483f4: h=malloc(...)
 80483f5: for (i=...)
 80483f6:   x = packed_code[i];
 80483f7:   h[i] = decrypt(x);
 80483f7: jmp *h
```

Malicious Code Analysis

- Packed code (*dynamic unpacking*)


0xc0000000 – 0xffffffff: kernel memory
0x00000000 – 0xbfffffff: user memory
.bss
.code
heap
stack

```
7a3e8018efa8aca8288
27281a82ef9a01ab181
1020a9a3bc9e99ff121
```

```
push %ebp
sub  $0x14, %esp
xor  %eax, %eax
...
```

```
.code

08048300 <decrypt>:
 8048300: ...

080483f4 <main>:
 80483f4: h=malloc(...)
 80483f5: for (i=...)
 80483f6:   x = packed_code[i];
 80483f7:   h[i] = decrypt(x);
 80483f7: jmp *h
```

Malicious Code Analysis

- Dynamic analysis
  - code is executed
  - sees instructions that are actually executed
- Problems of dynamic analysis
  - single path (execution trace) is examined
  - analysis environment possibly not *invisible*
  - analysis environment possibly not *comprehensive*
- Possible analysis environments
  - instrument program
  - instrument operating system
  - instrument hardware

Malicious Code Analysis

- Instrument program
  - analysis operates in same address space as sample
  - manual analysis with debugger
  - Detours (Windows API hooking mechanism)

  - binary under analysis is modified
    - breakpoints are inserted
    - functions are rewritten
    - debug registers are used
  - not invisible, malware can detect analysis
  - can cause significant manual effort

Malicious Code Analysis

- Instrument operating system
  - analysis operates in OS where sample is run
  - Windows system call hooks

  - invisible to (user-mode) malware
  - can cause problems when malware runs in OS kernel
  - limited visibility of activity inside program
    - cannot set function breakpoints

Malicious Code Analysis

- Instrument hardware
  - provide virtual hardware (processor) where sample can execute (sometimes including OS)
  - software emulation of executed instructions
  - analysis observes activity “from the outside”

  - completely transparent to sample (and guest OS)
  - operating system environment needs to be provided

Analysis Report

- File activity
  - read, write, create, open, …
- Registry activity
- Service activity
  - start or stop of Windows services (via Service Manager)
- Process activity
  - start, terminate process, inter-process communication
- Network activity
  - API calls and packet (network) logs

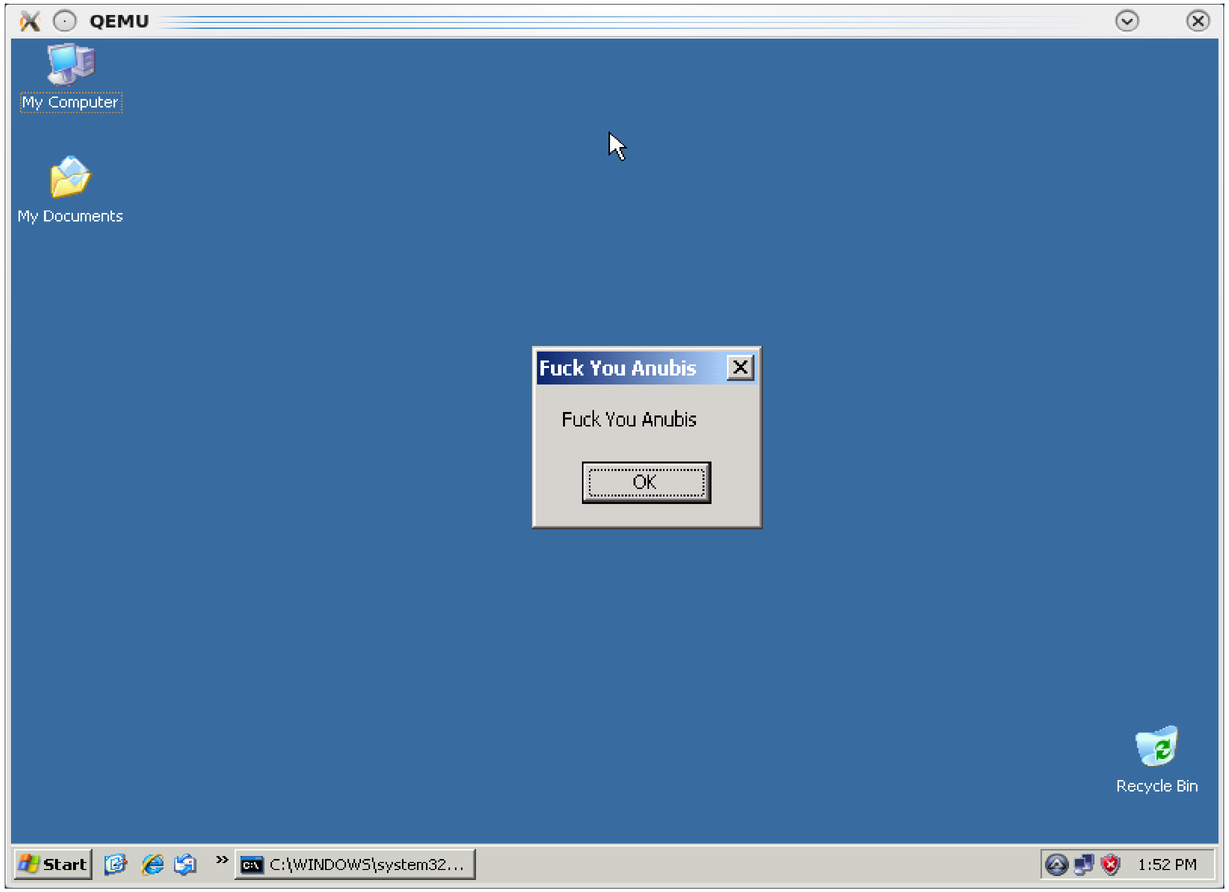
Stealth

- Virtual machines
  - allow to quickly restore analysis environment
  - identical, clean environment for every analysis run
  - introduces detectable artifacts

- Some detection mechanisms (we have seen)
  - x86 virtualization problems
  - speed of execution
  - check system/installation specific settings
  - computer name, drive label, external IP address, etc.

Stealth

```
$ ./analyze.py --show-window ~/anti_anubis.exe
```



Fuck You Anubis

Fuck You Anubis

OK

Overcomming
Anti-*

Anti Disassembly

- Running the binary should still work

- Try different disassembly methods / tools
- Help the disassembler to analyse the code
  - NOP out junk data
    - 0x90 → NOP
  - Remove some instructions (beware to not break intended functionality)
  - Connect pieces with unconditional jumps
    - If you can identify jump targets for indirect jumps
    - EB xx → JMP +xx

Patching

- Use a hex editor (hexedit)
- GDB
  - `gdb` (start gdb without a command to debug)

    ```
    (gdb) set write on
    (gdb) exec-file <progname>
    ```

    - File needs to be selected after write is set to on

    ```
    (gdb) set *0x4025a6=0xcc
    ```
- radare2
  - `oo+` (re-open file in write mode)

    `w 0x90` (write 0x90 at current possition)

Anti Debugging

- Reduce visibility of the debugger
- Use the appropriate breakpoint technique
- Intercept certain API functions to return fake results
  - Or patch jumps inside the binary
  - e.g JE (0x74) → JNE (0x75)

- Single step through problematic part manually and disable anti-debugging checks
  - Or script the process
  - Some tools also have functionality to work around certain checks

LD_PRELOAD

- Arguments for Dynamic Linker
  - Preloads given library before all other libraries
  - Can replace API calls

    e.g ptrace

- Can also be usefull to introduce determinism
  - e.g. replace calls to random or gettime with deterministic values to get the same results while debugging/analysing a binary

Anti-VM

- Try to change the execution environment
  - Run on a different VM
  - Tweak environment to avoid detection
  - Run on bare metal (beware!)
- Check what the binary reads/compares/executes to find anti-vm tricks
- Change control flow with a debugger
- Patch the binary to remove/avoid the checks

Summary

- Software reverse engineering
  - static & dynamic techniques

- Static techniques
  - check for strings, symbols, and library functions
  - disassembler

- Dynamic techniques
  - system/API call monitoring (`ptrace/ltrace` interface)
  - monitor network and file system activity
  - debugger

- Malicious code analysis