Proactive Computer Security

Software Reverse Engineering

Morten Brøns-Pedersen
<f@ntast.dk>

(Slides mostly by Morten Shearman Kirkegaard <moki@fabletech.com>)

2016-05-09

The art of taking things apart.

Why?

- To make compatible software
- To see how something works
- To find vulnerabilities

- Executable and Linkable Format (ELF) — Linux, BSD, MINIX, Solaris, IRIX, HP-UX, BeOS, GNU/Hurd, PlayStation, Wii, and a *lot* of embedded systems.
- Portable Executable (PE) — Windows
- Mach-O — Mach, OS X, iOS

What are we dealing with?

```
$ file something.bin
something.bin: ELF 32-bit LSB executable, Intel 80386,
version 1 (SYSV), dynamically linked (uses shared libs),
for GNU/Linux 2.6.26,
BuildID[sha1]=0x1fdc7c28e708e92a297e67448d8a9592bd12b02a,
not stripped
```

```
$ readelf -a something.bin |less
ELF Header:
  Magic:   7f 45 4c 46 01 01 01 00 00 00 00 00 00 00 00 00
  Class:                       ELF32
  Data:                        2's complement, little endian
  Version:                     1 (current)
  OS/ABI:                      UNIX - System V
  ABI Version:                 0
  Type:                        EXEC (Executable file)
  Machine:                     Intel 80386
  Version:                     0x1
  Entry point address:         0x8048320
...
```

```
$ readelf -a something.bin |less
...
Dynamic section at offset 0x598 contains 25 entries:
 Tag        Type              Name/Value
 0x00000001 (NEEDED)          Shared library: [libc.so.6]
 0x0000000c (INIT)            0x80482b8
 0x0000000d (FINI)            0x80484e0
 0x00000019 (INIT_ARRAY)      0x804958c
 0x0000001b (INIT_ARRAYSZ)    4 (bytes)
 0x0000001a (FINI_ARRAY)      0x8049590
 0x0000001c (FINI_ARRAYSZ)    4 (bytes)
 0x00000004 (HASH)            0x804818c
 0x6ffffef5 (GNU_HASH)        0x80481b4
 0x00000005 (STRTAB)          0x8048224
...
```

```
$ readelf -a something.bin |less
...
Symbol table '.dynsym' contains 5 entries:
Num:    Value Size Type    Bind   Vis      Ndx Name
  0: 00000000     0 NOTYPE LOCAL  DEFAULT UND
  1: 00000000     0 FUNC   GLOBAL DEFAULT UND puts@GLIBC_2.0
  2: 00000000     0 NOTYPE WEAK   DEFAULT UND __gmon_start__
  3: 00000000     0 FUNC   GLOBAL DEFAULT UND __libc_start_m
  4: 080484fc     4 OBJECT GLOBAL DEFAULT  16 _IO_stdin_used
...
```

Interesting Sections

- .text — Executable code
- .rodata — Read-only data
- .data — Initialized (read-write) data
- .bss — Uninitialized[1] (read-write) data

[1] Actually zero-initialized

After linking an executable, the section information can be stripped away.

The *program headers* will still give us the information we need.

```
$ readelf -a something.bin |less
...
Program Headers:
 Type      Offset   VirtAddr   FileSiz MemSiz  Flg Align
 PHDR      0x000034 0x08048034 0x00100 0x00100 R E 0x4
 INTERP    0x000134 0x08048134 0x00013 0x00013 R   0x1
  [Requesting program interpreter: /lib/ld-linux.so.2]
 LOAD      0x000000 0x08048000 0x0058c 0x0058c R E 0x1000
 LOAD      0x00058c 0x0804958c 0x00120 0x00124 RW  0x1000
 DYNAMIC   0x000598 0x08049598 0x000f0 0x000f0 RW  0x4
...
```

R = readable, **W** = writable, **E** = executable

PhysAddr (not shown here) is the same as VirtAddr on modern systems.

To make code compiled using different tool chains compatible, we need calling conventions.

Floating point numbers and structures are more complex, so we will ignore them in this lecture.

cdecl

Arguments are pushed to the stack. The last argument is pushed first. The *caller* removes the arguments from the stack, after the call.

On 80386 the result is stored in `EAX` (32 bit or less) or `EDX:EAX` (up to 64 bits).

The registers `EAX`, `ECX`, and `EDX` are *caller saved*, the rest are *callee saved*.

```
void my_func(int a, int b, int c);

my_func(1, 2, 3);

push 3
push 2
push 1
call my_func
add esp, 12
```

Question

Why do we push the last argument first?

Some functions have a variable number of arguments. The `printf()` function is a classic example.

```
int printf(const char *format, ...);
```

```
printf("i is %d, j is %d.\n", i, j);
```

Newer revisions of this convention requires the stack to be 16 byte aligned when entering a function. The alignment is ensured by `main()` and called functions maintain it.

```
0804848f <main>:
 804848f:     8d 4c 24 04     lea     ecx,[esp+0x4]
 8048493:     83 e4 f0        and     esp,0xfffffff0
 8048496:     ff 71 fc        push    DWORD PTR [ecx-0x4]
```

```
void my_func(int a, int b, int c);

my_func(1, 2, 3);

mov [esp+8], 3
mov [esp+4], 2
mov [esp], 1
call my_func
```

stdcall

The *"Standard" Calling Convention* is the calling convention used by Microsoft for Win32.

Much like *cdecl*; arguments are pushed to the stack, the last argument is pushed first.

But: The *callee* removes the arguments from the stack, when returning.

```
void my_func(int a, int b, int c);

my_func(1, 2, 3);

push 3
push 2
push 1
call my_func
```

(One Variation of) fastcall

The first two arguments are stored in `ECX` and `EDX`, the rest are pushed to the stack. The last argument is pushed first.

The *callee* removes the arguments from the stack, when returning.

```
void my_func(int a, int b, int c);

my_func(1, 2, 3);

push 3
mov edx, 2
mov ecx, 1
call my_func
```

Question

Why do we prefer to use registers instead of the stack?

Dead Listing (Static Analysis)

We look at the code. We do not execute it.

```
$ strings something.bin
/lib/ld-linux.so.2
M?Sn
__gmon_start__
libc.so.6
_IO_stdin_used
puts
__libc_start_main
GLIBC_2.0
PTRh0
Hello, World!
;*2$"
```

```
$ strings something.bin
/lib/ld-linux.so.2        ← loader/dynamic linker
M?Sn
__gmon_start__
libc.so.6
_IO_stdin_used
puts
__libc_start_main
GLIBC_2.0
PTRh0
Hello, World!
;*2$"
```

```
$ strings something.bin
/lib/ld-linux.so.2        ← loader/dynamic linker
M?Sn
__gmon_start__
libc.so.6                 ← the c library
_IO_stdin_used
puts
__libc_start_main
GLIBC_2.0
PTRh0
Hello, World!
;*2$"
```

```
$ strings something.bin
/lib/ld-linux.so.2          ← loader/dynamic linker
M?Sn
__gmon_start__
libc.so.6                   ← the c library
_IO_stdin_used
puts                        ← imported symbol
__libc_start_main
GLIBC_2.0
PTRh0
Hello, World!
;*2$"
```

```
$ strings something.bin
/lib/ld-linux.so.2         ← loader/dynamic linker
M?Sn
__gmon_start__
libc.so.6                  ← the c library
_IO_stdin_used
puts                       ← imported symbol
__libc_start_main
GLIBC_2.0
PTRh0
Hello, World!              ← interesting string
;*2$"
```

```
$ readelf -x .rodata ./something.bin

Hex dump of section '.rodata':
  0x080484f8 03000000 01000200  ........
  0x08048500 48656c6c 6f2c2057  Hello, W
  0x08048508 6f726c64 2100      orld!.


$ readelf -x .data ./something.bin

Hex dump of section '.data':
  0x080496a4 00000000 00000000  ........
```

```
$ objdump -M intel -d something.bin >something.txt
$ ~~vim~~ emacs something.txt
...
804840c: 55           push   ebp
804840d: 89 e5        mov    ebp,esp
804840f: 83 e4 f... and    esp,0xfffffff0
8048412: 83 ec 2... sub    esp,0x20
8048415: 83 7d 0... cmp    DWORD PTR [ebp+0x8],0x1
...
```

When writing programs in C, you may get the impression that execution of your program begins at `main()`. It doesn't.

The compiler will add some code (usually called `_start()`) to initialize your environment. You don't have to know much about this process.

The most important thing is the call to `__libc_start_main()`. The first argument to this call is a pointer to `main()`.

If you have symbols, great.

If not, you can still find `main()`, via the call to `__libc_start_main()`.

Unless the code is malicious—or deliberately obfuscated—you can skip ahead, and begin reverse engineering there. The startup code is not interesting.

Remember the ELF header?

```
$ readelf -a something.bin |less
ELF Header:
...
  Entry point address:    0x8048320
...
```

```
8048320:   31 ed        xor  ebp,ebp
8048322:   5e           pop  esi
8048323:   89 e1        mov  ecx,esp
8048325:   83 e4 f...   and  esp,0xfffffff0
8048328:   50           push eax
8048329:   54           push esp
804832a:   52           push edx
804832b:   68 70 8...   push 0x8048470
8048330:   68 80 8...   push 0x8048480
8048335:   51           push ecx
8048336:   56           push esi
8048337:   68 0c 8...   push 0x804840c <-- THIS MUST BE MAIN
804833c:   e8 cf f...   call 8048310 <__libc_start_main@plt>
8048341:   f4           hlt
```

```
804840c:  55         push ebp
804840d:  89 e5      mov  ebp,esp
804840f:  83 e4 f... and  esp,0xfffffff0
8048412:  83 ec 2... sub  esp,0x20
8048415:  83 7d 0... cmp  DWORD PTR [ebp+0x8],0x1
8048419:  75 0e      jne  8048429 <main+0x1d>
804841b:  c7 04 2... mov  DWORD PTR [esp],0x8048500
8048422:  e8 c9 f... call 80482f0 <puts@plt>
8048427:  eb 32      jmp  804845b <main+0x4f>
8048429:  c7 44 2... mov  DWORD PTR [esp+0x1c],0x0
```

So, we've found `main()`. How do we figure out what the code does?

We'll begin by looking at how common things in C look in assembly code.

Function Prologue

```
push    ebp
mov     ebp,esp
sub     esp,0x10
```

Function Epilogue

```
mov esp, ebp
pop ebp
ret
```

or

```
leave     (= mov esp, ebp ; pop ebp)
ret       (= pop ip)
```

Padding between Functions

```
90                     nop
66 90                  xchg   ax,ax
8d b4 26 00 00 00 00   lea    esi,[esi+eiz*1+0x0]
8d bc 27 00 00 00 00   lea    edi,[edi+eiz*1+0x0]
```

Arguments and Local Variables

Arguments are at `EBP` *plus* something:

```
mov     eax,DWORD PTR [ebp+0xc]
mov     edx,DWORD PTR [ebp+0x8]
```

Local variables are at `EBP` *minus* something:

```
mov     DWORD PTR [ebp-0x4],eax
```

The compiler is free to *not* use `EBP`, and just use offsets from `ESP`.

Pen and paper are your friends. Use them to draw stack frames.

Drawing exercise, yay!

Draw the stack layout for this program. Discuss with your peers.

```
unsigned int foo(unsigned int n, int x) {
  unsigned int a, b;
  if (n <= 1) {
    return x;
  }
  a = foo(n - 1, x);
  b = foo(n - 2, x);
  return a + b + x;
}
```

```
x = 123;
if (a == b) {
        puts("yes");
}
y = 456;
```

```
; ------------------------------
        mov [ebp-16], 123
        mov eax, [ebp-4]
        mov edx, [ebp-8]
        cmp eax, edx
        jnz .l1
; ------------------------------
        mov [esp], 0x8048510
        call 80482f0 <puts@plt>
; ------------------------------
.l1:
        mov [ebp-12], 456
; ------------------------------
```

```
x = 123;
while (i != 0) {
        puts("i is still != 0");
        i--;
}
y = 456;
```

```
; -------------------------------
        mov [ebp-16], 123
        mov ebx, [ebp-4]
; -------------------------------
.l0:
        test ebx, ebx
        jz .l1
; -------------------------------
        mov [esp], 0x8048510
        call 80482f0 <puts@plt>
        dec ebx
        jmp .l0
; -------------------------------
.l1:
        mov [ebp-12], 456
; -------------------------------
```

```
x = 123;
do {
        puts("some text");
        i--;
} while (i != 0);
y = 456;
```

```
; -------------------------------
        mov [ebp-16], 123
        mov ebx, [ebp-4]
; -------------------------------
.l0:
        mov [esp], 0x8048510
        call 80482f0 <puts@plt>
        dec ebx
        test ebx, ebx
        jnz .l0
; -------------------------------
        mov [ebp-12], 456
; -------------------------------
```

```
x = 123;
for (i=0; i<10; i++) {
        puts("some text");
}
y = 456;
```

```
; --------------------------------
        mov [ebp-16], 123
        xor ebx, ebx
; --------------------------------
.l0:
        cmp ebx, 10
        jge l1
; --------------------------------
        mov [esp], 0x8048510
        call 80482f0 <puts@plt>
        inc ebx
        jmp .l0
; --------------------------------
.l1:
        mov [ebp-12], 456
; --------------------------------
```

(Trick) Question

How can we tell if it's a `for` loop or a `while` loop?

Question

What does this function do?

```
80483dc: 55          push   ebp
80483dd: 89 e5       mov    ebp,esp
80483df: 83 ec 10    sub    esp,0x10
80483e2: 8b 45 0c    mov    eax,DWORD PTR [ebp+0xc]
80483e5: 8b 55 08    mov    edx,DWORD PTR [ebp+0x8]
80483e8: 01 d0       add    eax,edx
80483ea: 89 45 fc    mov    DWORD PTR [ebp-0x4],eax
80483ed: 8b 45 fc    mov    eax,DWORD PTR [ebp-0x4]
80483f0: c9          leave
80483f1: c3          ret
```

Coffee Break

```
int arr[3] = { 1, 2, 3 };
printf("%d\n", arr[1]);

mov    DWORD PTR [esp+0x1c],0x1
mov    DWORD PTR [esp+0x20],0x2
mov    DWORD PTR [esp+0x24],0x3

mov    eax,DWORD PTR [esp+0x20]

mov    DWORD PTR [esp+0x4],eax
mov    DWORD PTR [esp],0x8048530
call   8048300 <printf@plt>
```

(Trick) Question

How can we tell that it's an array?

```
int arr[3] = { 1, 2, 3 };
my_func(arr);

mov     DWORD PTR [esp+0x14],0x1
mov     DWORD PTR [esp+0x18],0x2
mov     DWORD PTR [esp+0x1c],0x3

lea     eax,[esp+0x14]

mov     DWORD PTR [esp],eax
call    804841c <my_func>
```

```
void my_func(int *arr)
{
        printf("%d\n", arr[2]);
}
```

```
...
mov     eax,DWORD PTR [ebp+0x8]
add     eax,0x8
mov     eax,DWORD PTR [eax]

mov     DWORD PTR [esp+0x4],eax
mov     DWORD PTR [esp],0x8048500
call    8048300 <printf@plt>
...
```

```
struct my_struct {
        int a;
        int b;
        int c;
};
struct my_struct x = { 1, 2, 3 };

mov     DWORD PTR [esp+0x14],0x1
mov     DWORD PTR [esp+0x18],0x2
mov     DWORD PTR [esp+0x1c],0x3
```

```
printf("%d\n", x.b);
```

```
mov     eax,DWORD PTR [esp+0x18]

mov     DWORD PTR [esp+0x4],eax
mov     DWORD PTR [esp],0x8048510
call    8048300 <printf@plt>
```

(Trick) Question

How can we tell that it's a structure?

```
struct my_struct x = { 1, 2, 3 };
my_func(&x);

mov    DWORD PTR [esp+0x14],0x1
mov    DWORD PTR [esp+0x18],0x2
mov    DWORD PTR [esp+0x1c],0x3

lea    eax,[esp+0x14]

mov    DWORD PTR [esp],eax
call   804841c <my_func>
```

```
void my_func(struct my_struct *ms)
{
        printf("%d\n", ms->b);
}
```

```
mov    eax,DWORD PTR [ebp+0x8]

mov    eax,DWORD PTR [eax+0x4]

mov    DWORD PTR [esp+0x4],eax
mov    DWORD PTR [esp],0x8048500
call   8048300 <printf@plt>
```

Usually—but not always—elements of structures are accessed as `pointer + constant`, and elements of arrays are accessed as `pointer + variable`.

If you execute my code, I control your computer!

Why do it, then?

It *can* be faster because:

- You can make fewer assumptions
- You can test your hypotheses
- It's easier to concentrate on code that matters

You can use everything you know from static analysis, but now you have data.

If you have symbols, gdb can place breakpoints based on function names.

```
$ gdb ./something.bin

(gdb) break main
Breakpoint 1 at 0x804840f

(gdb) run
Starting program: ./something.bin

Breakpoint 1, 0x0804840f in main ()
(gdb)
```

If you have symbols, gdb also knows which instructions belong to a function.

```
Breakpoint 1, 0x0804840f in main ()
(gdb) disassemble
Dump of assembler code for function main:
   0x0804840c <+0>:     push   ebp
   0x0804840d <+1>:     mov    ebp,esp
=> 0x0804840f <+3>:     and    esp,0xfffffff0
   0x08048412 <+6>:     sub    esp,0x20
   0x08048415 <+9>:     cmp    DWORD PTR [ebp+0x8],0x1
   0x08048419 <+13>:    jne    0x8048429 <main+29>
...
   0x0804845b <+79>:    mov    eax,0x0
   0x08048460 <+84>:    leave
   0x08048461 <+85>:    ret
End of assembler dump.
```

If you don't have symbols, gdb has no idea where a function begins or ends.

```
(gdb) break main
Function "main" not defined.
Make breakpoint pending on future shared library
load? (y or [n])
```

But you can still break on symbols imported from libraries.

```
(gdb) break __libc_start_main
Breakpoint 1 at 0x8048310

(gdb) run
Starting program: ./something.bin

Breakpoint 1, __libc_start_main (main=0x804840c, argc=1,
    ubp_av=0xffffd0b4, init=0x8048480, fini=0x8048470,
    rtld_fini=0xf7fee590, stack_end=0xffffd0ac)
    at libc-start.c:96
96      libc-start.c: No such file or directory.
```

If you have debugging info for your libc, gdb will know about function arguments.

```
Breakpoint 1, __libc_start_main (main=0x804840c, argc=1,
    ubp_av=0xffffd0b4, init=0x8048480, fini=0x8048470,
    rtld_fini=0xf7fee590, stack_end=0xffffd0ac)
    at libc-start.c:96
96      libc-start.c: No such file or directory.

(gdb) break *main
Breakpoint 2 at 0x804840c
```

Otherwise we'll just examine the stack. We know that `ESP` points to the return address, and that the first argument (at `ESP+4`) is a pointer to `main()`.

```
(gdb) x/2wx $esp
0xffffd08c:     0x08048341      0x0804840c

(gdb) break *0x0804840c
Breakpoint 2 at 0x804840c
```

If we are looking for vulnerabilities, we may care about *how* a program does something. Otherwise we only care about *what* it does.

A good reverse engineer will know what to reverse and what to skip.

What does `my_func()` do?

```
0x08048339 <+9>:     lea    eax,[esp+0x10]
0x0804833d <+13>:    mov    DWORD PTR [esp],eax
0x08048340 <+16>:    mov    DWORD PTR [esp+0x4],0x4d2
0x08048348 <+24>:    call   0x8048440 <my_func>
0x0804834d <+29>:    ...
```

```
(gdb) break *0x0804834d
Breakpoint 1 at 0x804834d

(gdb) run
Starting program: ./something.bin

Breakpoint 1, 0x0804834d in main ()
(gdb) x/12bx $esp+0x10
0xffffcff0:     0x31    0x32    0x33    0x34
0xffffcff4:     0x00    0xe5    0xfe    0xf7
0xffffcff8:     0x8b    0x84    0x04    0x08

(gdb) x/s $esp+0x10
0xffffcff0:      "1234"
```

Where is a function called from?

```
(gdb) break my_func
Breakpoint 1 at 0x8048440

(gdb) run
Starting program: ./something.bin

Breakpoint 1, 0x08048440 in my_func ()

(gdb) backtrace
#0  0x08048440 in my_func ()
#1  0x0804834d in main ()
```

Objects are just structures. If they have virtual methods, the first element of the structure is a pointer to an array of function pointers.

It's not easy to do static analysis of C++ programs.

You can learn a lot about how the compiler generates code, by compiling and debugging tiny programs. Do it.

There are plenty of reverse engineering challenges out there. Read other people's writeups, and solve some challenges.

In CTF competitions, they are often labeled *binary challenge*s. In other places they are called *crack-me*s and *keygen-me*s. Beware that some can be extremely difficult.

- `http://pwnable.kr`
- `http://treasure.pwnies.dk`

That's it. Have fun!